НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

____ _____ Теплоенергетичний факультет _____

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В.Коваль
(підпис) (ініціали, прізвище)
“___”_____2019 р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки 6.050101 “Комп’ютерні науки’’

на тему: Компонент рефакторингу в інтегрованому середовищі розробки Visual Studio

Виконала: студентка 4 курсу, групи ТР-51

Степанюк Анна Вікторівна _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к.т.н Тихоход В. О. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____
(підпис)

Київ – 2019

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки 6.050101 “Комп’ютерні науки’’

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2019 р.

ЗАВДАННЯ
на дипломну роботу студентці
Степанюк Анни Вікторівни
(прізвище, ім’я, по батькові)

1. Тема роботи “ Компонент рефакторингу в інтегрованому середовищі розробки Visual Studio”
керівник роботи доцент, к.т.н. Тихоход Володимир Олександрович
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)
затверджена наказом вищого навчального закладу від ”___”_____ 201__ р. № _____

2. Строк подання студентом роботи ___ _____ 201__ р.
3. Вихідні дані до роботи інсталятор з розширенням .vsix для компоненту рефакторингу
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) проаналізувати існуючі засоби автоматизованого рефакторингу та генератори програмного коду, удосконалити підхід до автоматизованого рефакторингу в IDE Visual Studio, розробити компонент для автоматизованого рефакторингу програмного коду в IDE Visual Studio.

_____
_____
_____
_____

5. Перелік ілюстраційного матеріалу (з точним зазначенням обов'язкових креслень) 1. Мета та завдання роботи 2. Огляд існуючих рішень 3. Переваги запропонованого рішення 4. Інструменти розробки 5. Класифікація функціоналу 6. Технологія використання 7. Результати роботи 8. Висновки

Дата видачі завдання "10" жовтня 2018 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Вивчення та аналіз задачі	14.10.2018-23.12.2018	
2.	Розробка архітектури та загальної структури системи	2.02.2019-3.03.2019	
3.	Розробка структур окремих підсистем	4.03.2019-14.04.2019	
4.	Підготовка матеріалів	15.04.2019-18.04.2019	
5.	Програмна реалізація системи	18.04.2019-14.05.2019	
6.	Захист програмного продукту	15.05.2019	
7.	Оформлення пояснювальної записки	16.05.2019-3.06.2019	
8.	Передзахист	28.05.2019	
9.	Захист	17.06.2019-22.06.2019	

Студентка _____ Степанюк А. В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Тихоход В. О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна записка містить опис рефалізації компоненту автоматизованого рефакторингу в IDE Visual Studio. Програмний продукт збільшує якість та продуктивність розробки через генерування нового коду, передбачення помилок, що можу виникнути на етапі виконання, використання оптимізованого та сучасного запису коду. Під час використання система ініціює процес рефакторингу, для блоку коду, вибраного користувачем. Компонент реалізовано з використанням платформи .Net компіляторів. Користувацький додаток представляє собою зручно встановлювану одиницю, яка розширює існуючий рефакторинг IDE Visual Studio.
Записка містить 67 сторінки, 24 рисунків та 14 посилань.

ABSTRACT

The diploma note contains a description of the refactoring of the automated refactoring component in IDE Visual Studio. The software product increases the quality and performance of the development by generating a new code, predicting errors that may occur during the execution phase, using an optimized and modern code entry. During use, the system initiates a refactoring process for a code block selected by the user. The component is implemented using the .Net compilers platform. The custom application is a comfortably installed unit that extends the existing refactoring of IDE Visual Studio.

The note contains 67 pages, 24 images and 14 references.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

IDE (англ. Integrated Developmen Environment) — комплексне програмне рішення для розробки програмного забезпечення;

API (англ. Application Programming Interface) — прикладний програмний інтерфейс;

SDK (англ. Software Development Kit) — набір із засобів розробки, утиліт і документації.

ВСТУП

З кожним роком підвищується складність програмних систем, що розробляються для промисловості. В сучасних швидкозмінних умовах важливим фактором успішності програмного продукту є використання еволюційної стратегії розробки та гнучких методик керування процесом. При такому підході з часом неминуче відбувається накопичення так званих “технічних боргів” [1], що негативно впливають на подальшу еволюцію програмного продукту. Тому невід’ємною складовою таких процесів є покращення якості коду за рахунок рефакторингу коду.

За визначенням Мартіна Фаулера рефакторинг (англ. refactoring) — процес зміни внутрішньої структури програми, що не впливає на її зовнішню поведінку та має на меті полегшити розуміння її роботи [2]. В основі рефакторингу лежить послідовність невеликих еквівалентних (що не змінюють поведінку) перетворень.

Важливим є підтримка даного процесу з IDE. Існує значна кількість таких засобів рефакторингу. Одними з найпопулярніших розширень для IDE Microsoft Visual Studio є ReSharper та CodeRush.

Проаналізовано основні недоліки існуючих засобів рефакторингу:

1. мають обмеження, оскільки це комерційні продукти;
2. мають значний вплив на продуктивність та швидкість роботи IDE;
3. не мають чіткої спеціалізації на рефакторинг, оскільки пропонують розширення та інструментарій інших компонентів IDE (синтаксичний аналізатор коду, інструмент для юніт тестуванн, редактора коду).

На основі отриманого аналізу було прийнято рішення розробити новий компонент для розширення функціоналу рефакторингу в IDE Visual Studio. Завдяки використанню API, що надає платформа компіляторів .Net, вплив на продуктивність IDE є мінімальною. Новий компонент рефакторингу в IDE Visual Studio є відкритим до загального користування продуктом.

1 ПОСТАНОВКА ЗАДАЧІ РОЗРОБКИ КОМПОНЕНТА РЕФАКТОРИНГУ В ІНТЕГРОВАНОМУ СЕРЕДОВИЩІ РОЗРОБКИ VISUAL STUDIO

Метою роботи є розробка компоненту для автоматизованого рефакторингу програмного коду в IDE Visual Studio.

Для досягнення мети були поставлені наступні завдання:

— проаналізувати існуючі засоби автоматизованого рефакторингу;

— проаналізувати існуючі реалізації засобів автоматизованого рефакторингу програмного коду;

— проаналізувати існуючі генератори програмного коду;

— дослідити існуючі метрики коду та їх актуальність;

— удосконалити підхід до автоматизації рефакторингу в IDE Visual Studio;

Вихідний продукт повинен представляти собою зручний і зрозумілий інструмент рефакторингу коду для мови C#. Компонент має бути розроблений у вигляді розширення (надбудови) для IDE Microsoft Visual Studio. Інтеграція розширення з середовищем розробки повинна відбуватись через інсталятор.

Рефакторинг вихідного коду повинен підвищити якість та підтримку коду проекту шляхом:

— передбачення помилок на етапі виконання;

— впровадження оптимізованого синтаксису мови програмування C#;

— генерація нового коду;

Цільовою аудиторією даного програмного продукту будуть користувачі IDE Microsoft Visual Studio, які програмують на мові C#.

Інструмент автоматизованого рефакторингу повинен бути легко встановлюваною одиницею, яка взаємодіє з IDE шляхом додавання нових пропозицій по рефакторінгу у меню “Швидкі дії” (“Quick Actions”), щоб надати користувачу можливість швидкого використання рефакторингу для коду над яким він працює.

Компонент рефакторингу повинен легко інтегруватись з іншими шаблонами компонентів, які створюють команди меню, вікна інструментів та розширення редактора.

Програмний продукт повинен аналізувати частину коду, в якій стоїть курсор та після відповідності необхідним правилам відображати запропонований рефакторинг. Основними функціями компоненту мають бути: генерація нового коду, трансформація коду до сучасної форми запису, усунення регулярних помилок, що виникають на етапі виконання.

2 АНАЛІЗ ЗАСОБІВ РЕФАКТОРИНГУ ТА АНАЛІЗУ КОДУ

2.1 Автоматизовані засоби рефакторингу

ReSharper є комерційним програмним продуктом для збільшення продуктивності роботи та автоматизації рефакторингу для Microsoft Visual Studio розроблений компанією JetBrains.

ReSharper забезпечує підтримку функціоналу для C#, VB.NET, XAML, JavaScript, TypeScript, XML, HTML, CSS, ASP.NET, ASP.NET MVC.

Додаток надає нові засоби автозаповнення, навігації, пошуку, виділення синтаксису, форматування, оптимізації та генерації коду, надає близько 60 автоматизованих рефакторингів, спрощує модульне тестування в середовищах MSTest та NUnit.

DevExpress CodeRush - це комерційний плагін для Visual Studio, який дозволяє розробнику швидше писати більш якісний код, налагоджувати його, запускати тести, виявляти дефекти і виконувати інші корисні функції.

CodeRush надає великий функціонал для юніт тестування. Додаток автоматично визначає модульні тести для фреймворків NUnit, xUnit, MSpec і MSTest, а також може одночасно запускати тести, розташовані на декількох збірках.

В доступі є функція Analyze Code Coverage, яка допомагає визначити частини вашого рішення які охоплюються модульними тестами, і знаходити місця ризику у вашій програмі.

Основною метою програмних продуктів ReSharper та CodeRush є покращення продуктивності розробників при написанні коду в середовищі розробки від Microsoft.

Вони проводять статичний аналіз коду і автоматично визначають такі проблеми, як помилки компіляції, логічні помилки, помилки на етапі виконання і

використання неоптимальних конструкцій, підсвічування потенційних проблем в вікні редактора коду.

Для багатьох з потенційних проблемних частин коду, інтегроване розширення пропонує варіанти автоматичного виправлення. Окрім того ReSharper пропонує набір різних функцій для роботи з конструкціями кода що дублюються. Для прикладу, генерація коду для типу може допомогти в створення конструкторів, властивостей, перевизначенню методів, делегатів і інших членів. Існуючі вбудовані шаблони покривають більшість типових конструкцій такого типу як if...else або try...catch, такоє є можливість створення власних користувацьких шаблонів.

У даних розширень є набір рефакторингу, який по кількості значно більше ніж ті, що пропонує за замовчуванням середовище розробки Visual Studio.

В додаток, плагіни надають функціонал спрощеної навігації, пошуку по коду і налаштування форматування, що дозволяють, для прикладу, змінювати заголовки файлів.

Недоліком вище описаних програмних продуктів для автоматизованого рефакторингу є те, що для користувача який потребує тільки компонент рефакторингу, встановлюються додаткові компоненти які розширюють редактор коду, модулі для модульного тестування з використанням різноманітних фреймворків. Великий спектр функціоналу який надають дані плагіни має безпосередній вплив на продуктивність IDE Visual Studio.

На основі існуючих ReSharper та CodeRush, запропоновано новий авоматизований компонент рефакторингу для мови C#. Компонент чітко сфокусований на автоматизованому рефакторингу. Перевагами такого рішення є те що середовище розробки не навантажується додатковими інструментами, та користувачу, який встановлює компонент рефакторингу, не нав’язується додатковий функціонал, який не має відношенння до рефакторинга. Розроблений інструмент буде

написаний на платформі Roslyn, використовуючи його API для аналізу коду та написання рефакторинг провайдерів.

2.2 Roslyn

Roslyn (рисунок 2.1) — платформа з відкритим вихідним кодом, що розробляється корпорацією Microsoft, і містить в собі компілятори і засоби для розбору і аналізу коду, написаного на мовах програмування C # і Visual Basic.

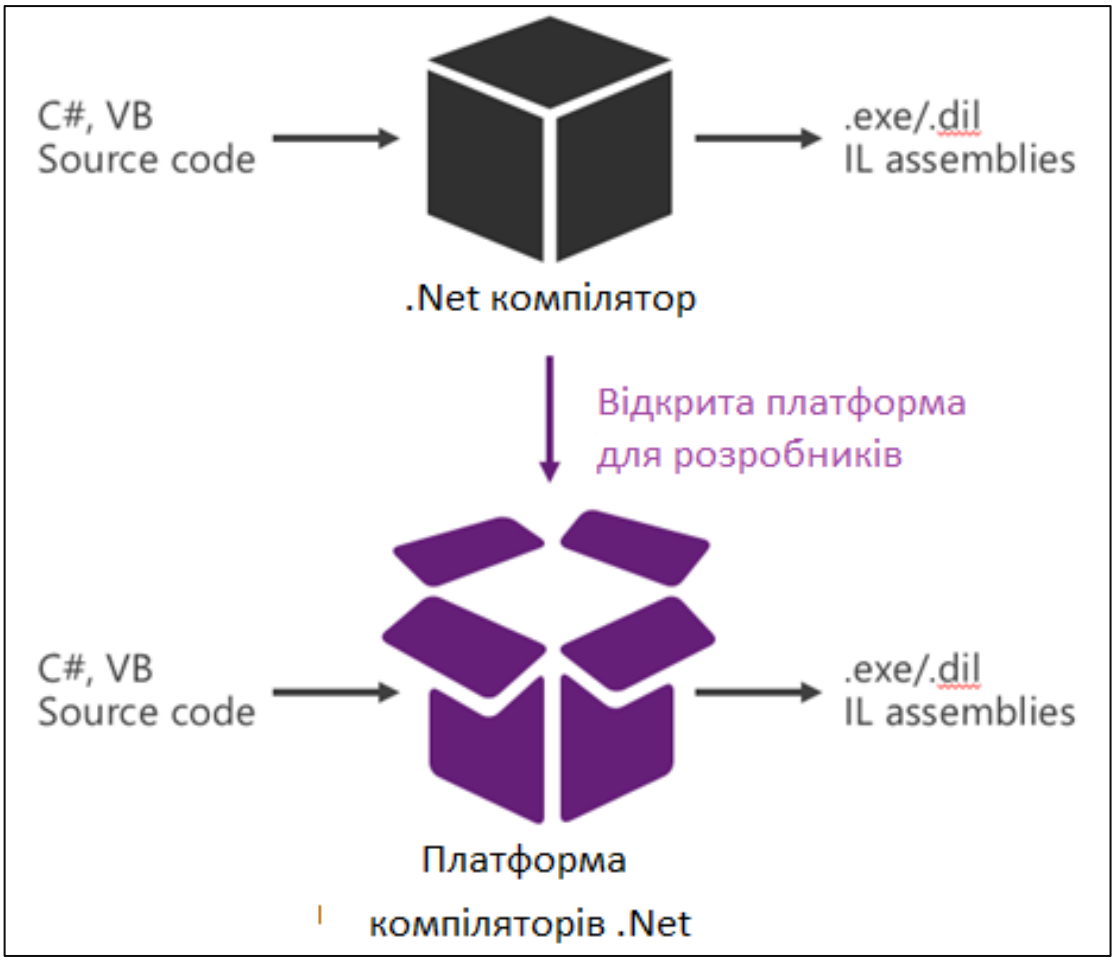

Рисунок 2.1 – Схема роботи компіляторів .Net

За допомогою засобів аналізу, наданими платформою Roslyn, можна проводити

повний розбір коду, аналізуючи всі підтримувані конструкції мови.

Платформа дозволяє створювати аналізатори та засоби виправлення коду для виявлення і усунення помилок. Аналізатори вивчають синтаксис і структуру коду, а також виявляють методи, які потрібно виправити. Засоби виправлення коду надають один або кілька рекомендованих способів усунення помилок в коді, виявлених аналізаторами.

Roslyn надає єдиний набір API-інтерфейсів, які дозволяють перевіряти і аналізувати базу коду C# або Visual Basic. Єдина база коду спрощує створення аналізаторів і засобів рефакторингу. При цьому використовуються API-інтерфейси, які дозволяють проводити синтаксичний і семантичний аналіз. Функції виконання комплексного аналізу належать компілятору в тей час як розробник може зосередитись на розробці автоматизованого рефакторінгу та більш поглиленого аналізу.

Додатковою перевагою є те, що аналізатори та засоби виправлення коду використовують набагато менше пам'яті при завантаженні в Visual Studio, ніж знадобилося б при написанні власної бази коду для аналізу коду в проекті. Використовуючи ті ж API, що компілятор і IDE, ви можете створювати власні інструменти статичного аналізу. Це означає, що члени вашої команди зможуть використовувати розширені аналізатори та засоби рефакторінгу без істотного впливу на продуктивність IDE.

Середовище Visual Studio дозволяє створювати на основі Roslyn як вбудовані в саму IDE інструменти (розширення Visual Studio), так і незалежні додатки (standalone інструменти).

Висновки до розділу

Проаналізувавши функціонал існуючих засобів автоматизованого рефакторингу, було виявлено недоліки в даних засобах:

1. мають обмеження, оскільки це комерційні продукти;
2. мають значний вплив на продуктивність та швидкість роботи IDE;
3. не мають чіткої спеціалізації на рефакторинг, оскільки пропонують розширення та інструментарій інших компонентів IDE (синтаксичний аналізатор коду, інструменти для юніт тестування та редактора коду).

На основі отриманого аналізу було прийнято рішення розробити новий компонент для розширення функціоналу рефакторингу в IDE Visual Studio. Завдяки використанню API, що надає платформа “Roslyn”, влив на продуктивність IDE є мінімальною. Новий компонент рефакторингу в IDE Visual Studio є відкритим до загального користування продуктом.

3 ЗАСОБИ РОЗРОБКИ КОМПОНЕНТА РЕФАКТОРІНГУ В IDE VISUAL STUDIO

Важливим завданням при розробці програмного продукту є вибір таких інструментів, які б полегшили роботу програміста, надавши всі необхідні інструменти для реалізації поставленої задачі, і дали змогу розробити програмний продукт, який повністю задовільняє вимогам.

При реалізації програмного забезпечення були використані такі засоби розробки:

— середовище розробки Visual Studio;

— мову програмування C#;

— пакету Visual Studio SDK;

— платформу компиляторів .NET (“Roslyn”) SDK;

— експерементальний екземпляр Visual Studio;

Важливим також є вибір операційної системи, на якій буде розроблятися програмний продукт. Виходячи з того, що операційна система Microsoft Windows є найпоширенішою, багатофункціональною, відповідає вимогам сучасних програмних застосунків, сумісна з іншими ОС, добре захищена, зручна і надійна, то було вибрано її середовище (а саме — Microsoft Windows 10).

Можливість використання програмного продукту не залежить від операційної системи. Це може бути як комп’ютер з ОС Windows, Mac чи UNIX з встановленою версією Visual Studio.

3.1 Середовище розробки Visual Studio

Для створення програмного проекту було використано платформу Windows 10

із встановленим середовищем розробки Visual Studio Community 2017.

Середовище розробки Visual Studio дозволяє швидко і ефективно писати код, незалежно від мови, від C #, VB і C ++ до JavaScript і Python, надаючи допомогу в реальному часі.

З кожною версією Visual Studio Microsoft враховує побажання розробників і вдосконалює інструменти розробки для створення додатків практично для будь-якої платформи. Результатом є величезний інтерес і більше 21 млн установок інструменту на сьогоднішній день

IntelliSense описує API під час введення, а автоматичне завершення збільшує швидкість і точність роботи. Знайомство з новим API прискорюється завдяки звуженню набору значень за категоріями. IntelliSense дозволяє перевіряти визначення API.

Архітектура розробки Visual Studio підтримує можливість використання доповнень (Add-Ins) — плагінів від сторонніх розробників, що дозволяє додавати нову функціональність практично на кожному рівні, включаючи додавання підтримки систем контролю версій вихідного коду, додавання нових наборів інструментів (наприклад, для рефакторінгу та синтаксичного аналізу коду) або інструментів для інших аспектів процесу розробки програмного забезпечення.

У зв'язку з тим, що середовище Visual Studio має тривалу історію і дозволяє реалізовувати різноманітні програмні продукти на своїй основі, з'явився термін Visual Studio Extensibility (VSX), який об'єднує в собі все, що відноситься до розширення функціоналу IDE.

3.2 Технологія для розробки розширення в IDE

Для створення add-in в IDE Visual Studio було встановлено пакет SDK що надає інструменти по розширенню функціоналу та комбінуванню нових можливостей в середовищі розробки. Нижче перераховані способи розширення Visual Studio:

— команди, кнопки, меню та інші елементи інтерфейсу користувача

— вікна засобів для нових функціональних можливостей

— розширення IntelliSense для існуючої мови та підтримка технології IntelliSense для нових мов програмування

— надання пропозицій по рефакторінгу у Quick Actions меню, що допомагають розробникам створювати більш якісний програмний продукт

— підтримка нової мови програмування

— додавання користувацького типу проекту

Visual Studio надає можливість розширити практично будь-яку частину: меню, панель інструментів, команди, вікна, рішення, проекти, редактори та інше.

Найбільшої популярності набирає розширення функцій команд, меню, панелей інструментів, вікон, IntelliSense і шаблони проектів.

Розширення меню і команд: додавати власні елементи в меню Visual Studio і на панелі інструментів. Їх можна використовувати для запуску нових функцій Visual Studio або власні зовнішні допоміжні програми. Є можливість налаштування комбінацій клавіш для елементів меню.

Редактор Visual Studio створюється за допомогою компонентів Managed Extensibility Framework (MEF). Можна створювати власні компоненти MEF для розширення редактора, а ваш код може використовувати редактор компонентів.

Платформа MEF - це бібліотека для створення простих розширюваних додатків. Вона дозволяє розробникам додатків знаходити і використовувати розширення без будь-яких налаштувань. Вона також дозволяє розробнику розширення легко інкапсулювати код і уникати ненадійних жорстких залежностей. MEF дозволяє повторно використовувати в додатках не тільки розширення, але й цілі програми.

Шаблон проекту VSIX можна використовувати для створення розширення або для упаковки існуючого розширення. Шаблон проекту VSIX підтрирмується для Visual Basic та C# мов. Даний шаблон встановлюється як частина пакету VSSDK.

Шаблон проекту VSIX складається тільки з source.extension.vsixmanifest файл, який містить відомості про розширення і ресурси, що випускається.

Конструктор маніфесту VSIX містить чотири розділи, які відповідають таким елементам верхнього рівня (рисунок 3.1):

— метадані

— цільові об'єкти установки

— ресурси

— залежності

Область заголовка містить наступні елементи:

— назва продукту, яка описує ім'я розширення.

— ідентифікатор продукту вказує унікальний код для пакета.

— ім'я автора модуля.

— номер версії розширення.

В вкладці метаданих є текстовий опис модуля, який буде відображатися в Диспетчері розширень. Атрибут мови відповідає мовній версії середовища виконання (CLR). Є можливість додати текстовий файл ліцензії. За бажанням можна додати графічний файл в форматі .png, .bmp, .jpeg, .ico який відображатиметься в Диспетчері розширень. Для зручної фільтрації розширення пропонується додати теги. Посібник для користування програми може містити файл формату .txt, .rtf, що повідомляє як використовувати функціонал розширення.

Вкладка “Install targets” містить інформацію про тип установки, що має два можливих значення Розширення Visyual Studio та Розширення SDK. Для першого варіанту слід вказати список продуктів Visual Studio для яких буде доступним розроблюване розширення. Кожний продукт окремо ідентифікується по імені та версії. Для розширення пакету SDK вказується глобальна установка, яка не обмежена певним продуктом і версією. Ідентифікатор цільової платформи - це Windows.

Вкладка “Assets” містить список, який описує елементи розширення або вміст, що цей пакет поверхні. Тип, а також шлях розширення або вміст елемента відповідає Type і Path атрибутам конкретного Asset елемента. В програмному продукті використовується Microsoft.VisualStudio.MefComponent.

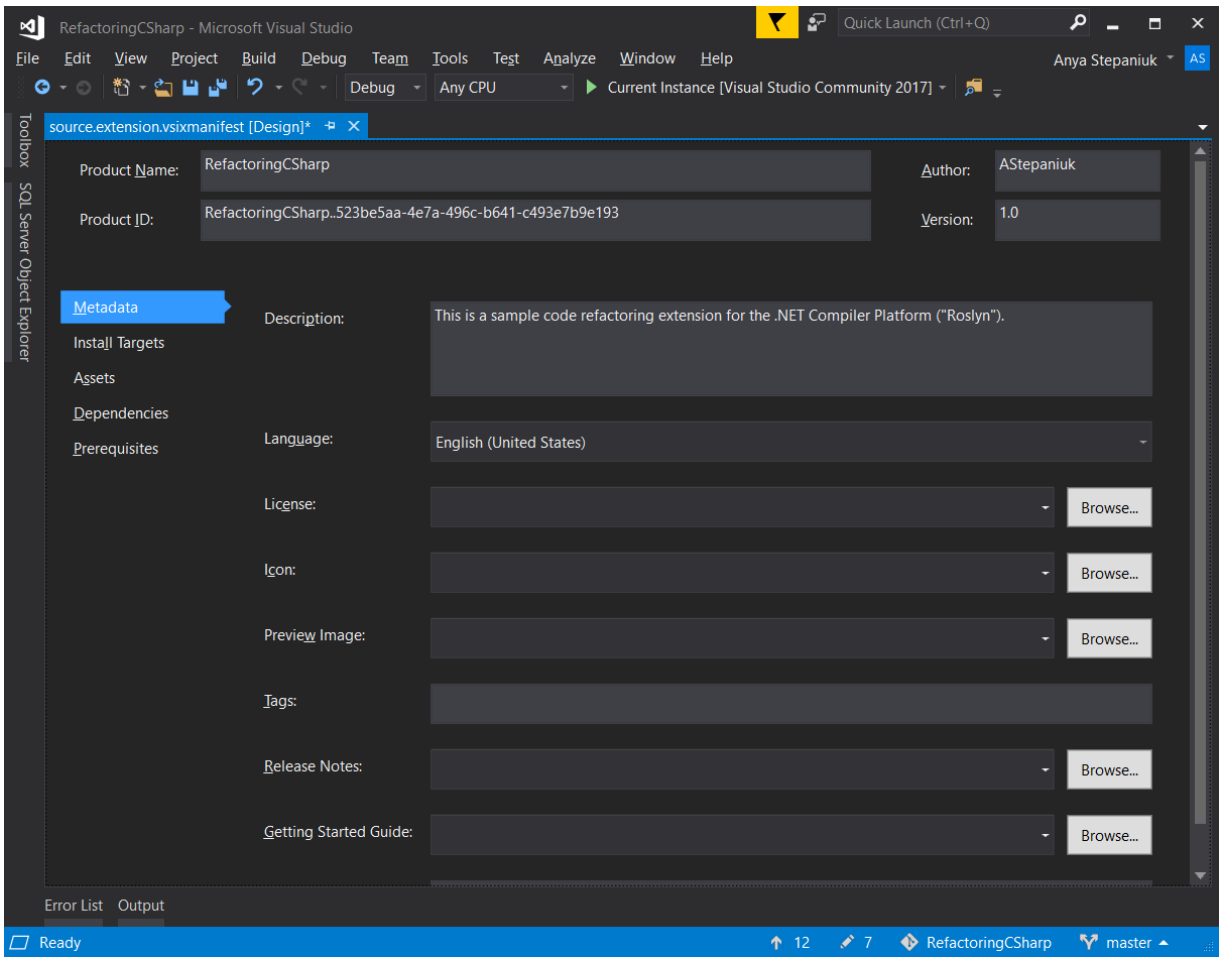

Рисунок 3.1 — Маніфест VSIX проекту

3.3 Технологія для розробки компоненту рефакторінгу

Компілятори створюють деталізовану модель коду програми для перевірки синтаксису і семантики цього коду. Пакет SDK для .NET Compiler Platform надає доступ до цієї моделі. Ми застосовуємо засоби рефакторінгу та аналізу коду, щоб покращити його якість, та генератори коду, щоб спростити розробку додатків.

Це основне призначення API-інтерфейсів Roslyn: відкривати "чорні ящики", щоб користувачі могли оперувати в розробці даними, використовуючи масу відомостей про код, доступних в компіляторі. На відміну від непрозорої роботи перетворювачів вихідного коду на вході в об'єктний код на виході, компілятори з

використанням Roslyn виконують функції платформ. Ці API-інтерфейси можна використовувати для розширення існуючих додатків та написання нових.

.NET Compiler Platform значно спрощує створення засобів і додатків, орієнтованих на код. Він надає безліч можливостей для інновацій в таких областях, як метапрограмування, створення і перетворення коду, інтерактивне використання мов C # і Visual Basic, а також їх впровадження в доменні мови.

Платформа дозволяє створювати аналізатори та засоби виправлення коду для виявлення і усунення помилок. Аналізатори вивчають синтаксис і структуру коду, а також виявляють методи, які потрібно виправити. Засоби виправлення коду надають один або кілька рекомендованих способів усунення помилок в коді, виявлених аналізаторами. Зазвичай аналізатор і відповідні засоби виправлення коду упаковані в один проект.

Для вивчення коду в аналізаторах і засобах виправлення використовується статичний аналіз. Вони не запускають код і не надають інші переваги тестування. Але вони можуть вказувати на методи, використання яких часто призводить до помилок, допомагають виявити підтримуваний код, а також перевіряють дотримання стандартних рекомендацій.

Висновки до розділу

Архітектура Visual Studio дозволяє легко інтегрувати нові компоненти. Було використано пакет SDK, що надає інструменти по розширенню функціоналу та комбінуванню нових можливостей в середовищі розробки.

Платформа “Roslyn” надає відкритий API для розробки нових методів рефакторингу. Перевагою даної платформи є те, що вона широко використовується в сучасній реалізації Visual Studio. Зважаючи на це, вплив на продуктивність та швидкість роботи є мінімальною.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ КОМПОНЕНТУ РЕФАКТОРІНГУ В IDE VISUAL STUDIO

4.1 Архітектура програмного рішення

Програмний компонент побудований з використанням шаблону проекту Code Refactoring (VSIX). Програмне рішення (рисунок 4.1) складається з проекту розширення RefactoringCSharp.vsix, що є стартовим проектом при запуску. Проект VSIX містить файл маніфест з необхідною інформацією, що стосується інсталяції розширення, основних відомостей щодо функціоналу та інше. Також маніфест має посилання на проект типу Class Library версії .Net Standard 1.3.

Бібліотека RefactoringCSharp.dll складається з великої кількості класів, такі що безпосередньо реалізують логіку рефакторингу, розширюють функціонал класів платформи компіляторів .Net, та інші допоміжні класи. Останні з використання об’єктно-орієнтованого підходу є гнучким та зрозумілим рішенням для підтримки існуючого рефакторингу та написанню нового. Рефакторинг провайдер безпосередньо є головною логічною одиницею, яка виконує логіку по аналізу потенційних місць в коді, для яких може бути передбачений рефакторинг.

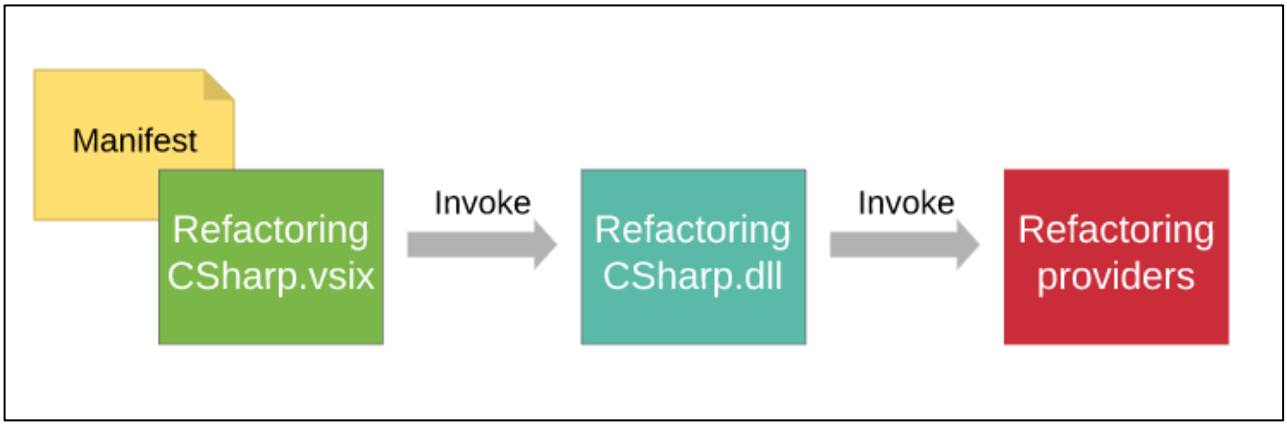

Рисунок 4.1 – Архітектура компоненту

4.2 Використання Roslyn

4.2.1 Опис роботи з аналізатором

Перед тим, як безпосередньо починати аналіз коду, необхідно отримати список файлів, вихідний код яких буде перевірятися, а також отримати сутності, необхідні для коректного аналізу. Можна виділити кілька пунктів, які потрібно виконати для отримання необхідних для аналізу даних (рисунок 4.1):

— створення workspace;

— отримання solution (опціонально);

— отримання проектів;

— розбір проекту: отримання компіляції, списку файлів;

— розбір файлу: отримання синтаксичного дерева і семантичної моделі;

Створення робочого простору (workspace) необхідно для отримання рішення або проектів. Для отримання workspace необхідно викликати статичний метод Create класу MSBuildWorkspace, який повертає об'єкт типу MSBuildWorkspace.

Отримання solution актуально, коли необхідно проаналізувати не один проект що входить в дане рішення. Тоді, отримавши solution, легко можна отримати список всіх вхідних в нього проектів.

Для отримання solution використовується метод OpenSolutionAsync об'єкта MSBuildWorkspace. У підсумку отримуємо колекцію, яка містить в собі список проектів (тобто об'єкт IEnumerable <Project>).

Якщо відсутня необхідність в аналізі всіх проектів, можна отримати конкретний проект, використовуючи асинхронний метод OpenProjectAsync об'єкта MSBuildWorkspace. Використовуючи цей метод, отримуємо об'єкт типу Project.

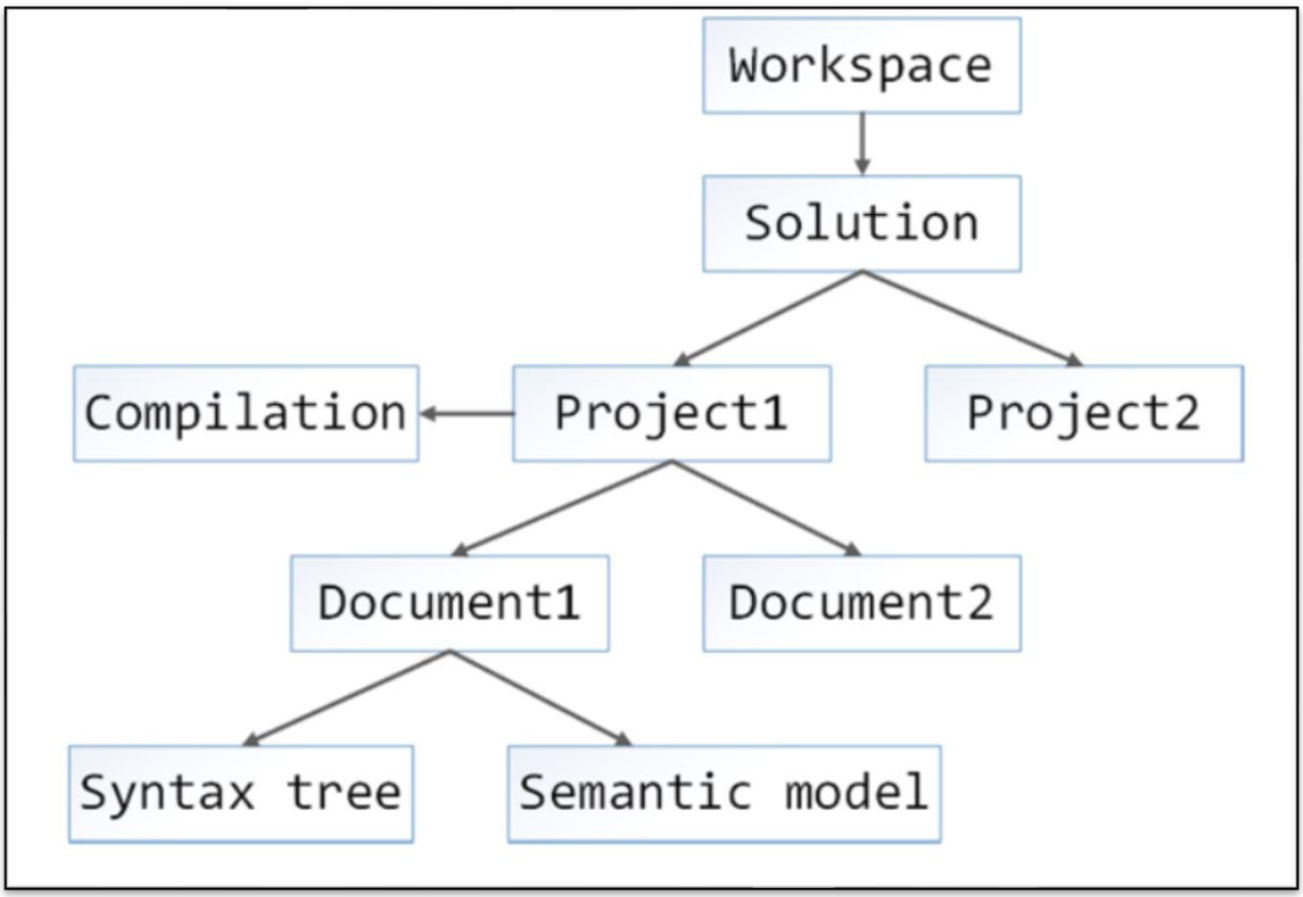

Рисунок 4.1 – Структура програмного рішення

Після того, як отримано список проектів для аналізу, можна приступати до їх розбору. Результатом розбору проекту повинен стати список файлів для аналізу і компіляція. Список файлів отримати просто - для цього використовується властивість Documents екземпляра класу Project. Для отримання компіляції використовується метод TryGetCompilation або GetCompilationAsync.

Отримання компіляції - один з ключових моментів, так як саме вона використовується для отримання семантичної моделі, необхідної для проведення глибокого і складного аналізу вихідного коду.

Наступний етап - розбір файлу. При ньому необхідно отримати дві сутності, на яких і базується повноцінний аналіз - синтаксичне дерево і семантичну модель. Синтаксичне дерево будується на підставі вихідного коду програми і використовується для аналізу різних конструкцій мови. Семантична модель надає інформацію про об'єкти і їх типи.

Для отримання синтаксичного дерева (об'єкт типу SyntaxTree) використовується метод TryGetSyntaxTree або GetSyntaxTreeAsync екземпляра класу Document.

Семантична модель (об'єкт типу SemanticModel) можна отримати з компіляції шляхом використанням синтаксичного дерева. Для цього використовується метод GetSemanticModel екземпляра класу Compilation, що приймає в якості обов'язкового параметра об'єкт типу SyntaxTree.

Синтаксичне дерево є базовим елементом, необхідним для аналізу коду. Саме по ньому відбувається переміщення в ході аналізу. Дерево будується на основі коду, наведеного в файлі, з чого випливає висновок, що кожен файл має своє синтаксичне дерево. Крім цього варто враховувати той факт, що синтаксичне дерево є незмінним. Тобто змінити його можна, викликавши відповідний метод, але результатом його роботи буде нове синтаксичне дерево.

4.2.2 Опис роботи з синтаксичною моделлю

Дерева синтаксису утворюють первинну структуру, що використовується для компіляції, аналізу коду, прив'язки, рефакторінга, функцій інтегрованого середовища розробки і створення коду.

Для зручного представлення дерева синтаксису “Roslyn” надає доступ до вікна інструментів “Візуалізатор синтаксису” (рисунок 4.2). Дане вікно дозволяє переглядати і аналізувати дерева синтаксису. Цей інструмент відіграє важливу роль, оскільки допомагає зрозуміти моделі кода, які слід проаналізувати.

Після отримання дерево являє собою знімок поточного стану коду і ніколи не змінюється. Це дозволяє декільком користувачам одночасно взаємодіяти з одним деревом синтаксису в різних потоках без блокування або дублювання. Дерева ефективно використовують базові вузли повторно, тому нову версію можна перебудувати швидко та з невеликими витратами пам'яті.

Дерево синтаксису (рисунок 4.3) фактично є деревовидної структурою даних, де нетермінальні структурні елементи є батьківськими для інших елементів. Кожне дерево синтаксису складається з вузлів, токенів і додаткової синтаксичної інформації (trivia).

Синтаксичні вузли є одним з основних елементів дерев синтаксису. Вони представляють такі синтаксичні конструкції, як оголошення, оператори, пропозиції та вирази. Кожна категорія синтаксичних вузлів представлена окремим класом, похідним від Microsoft.CodeAnalysis.SyntaxNode.

Всі синтаксичні вузли є нетермінальнимі вузлами в дереві синтаксису. Як дочірній елемент іншого вузла кожен вузол має батьківський вузол, до якого можна звернутися за допомогою властивості SyntaxNode.Parent.

Кожен вузол має метод SyntaxNode.ChildNodes(), який повертає список дочірніх вузлів в послідовному порядку з урахуванням їх позиції в тексті вихідного коду. Цей список не містить маркери.

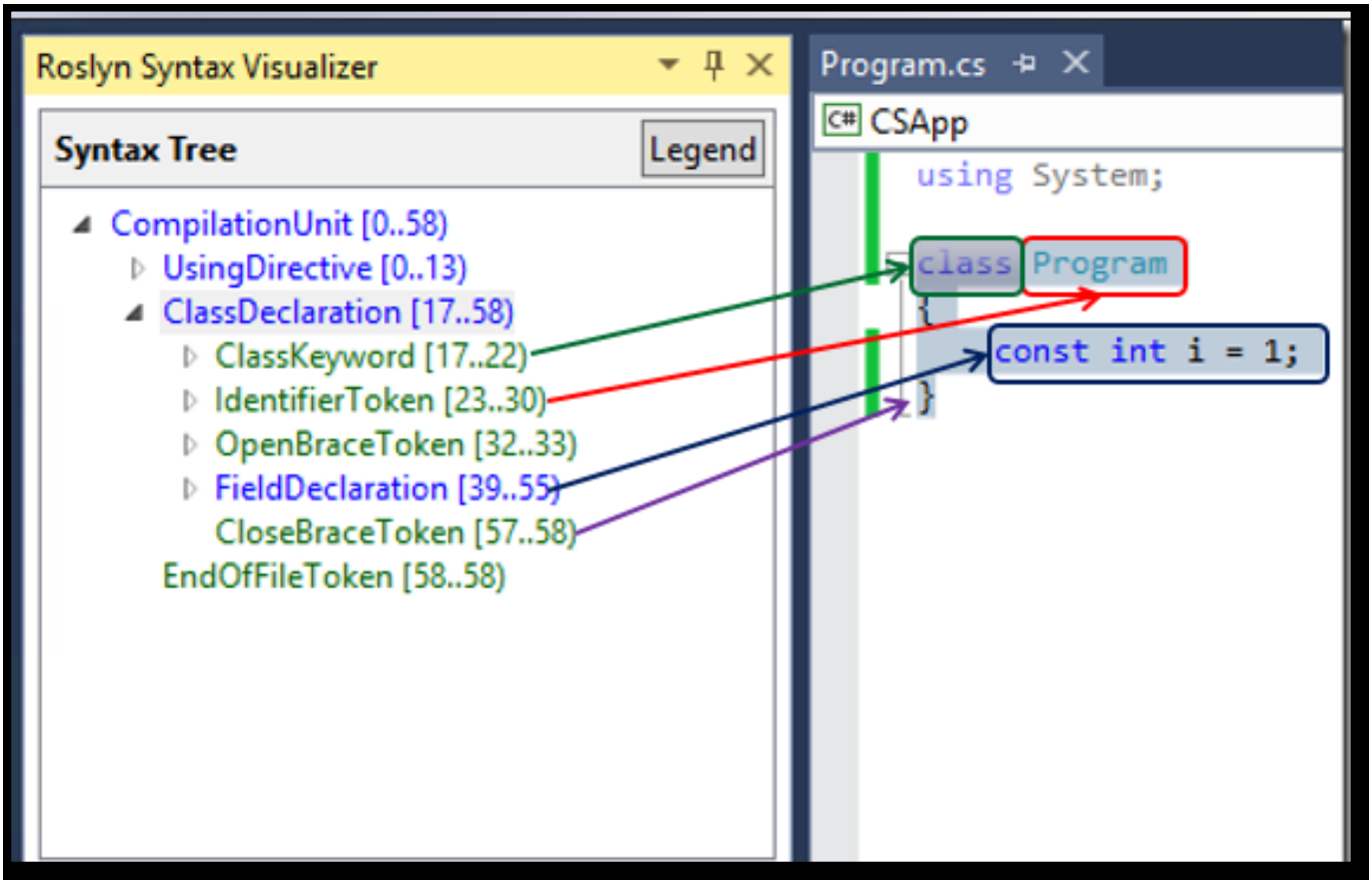

Рисунок 4.2 – Вікно “Синтаксичний візуалізатор”

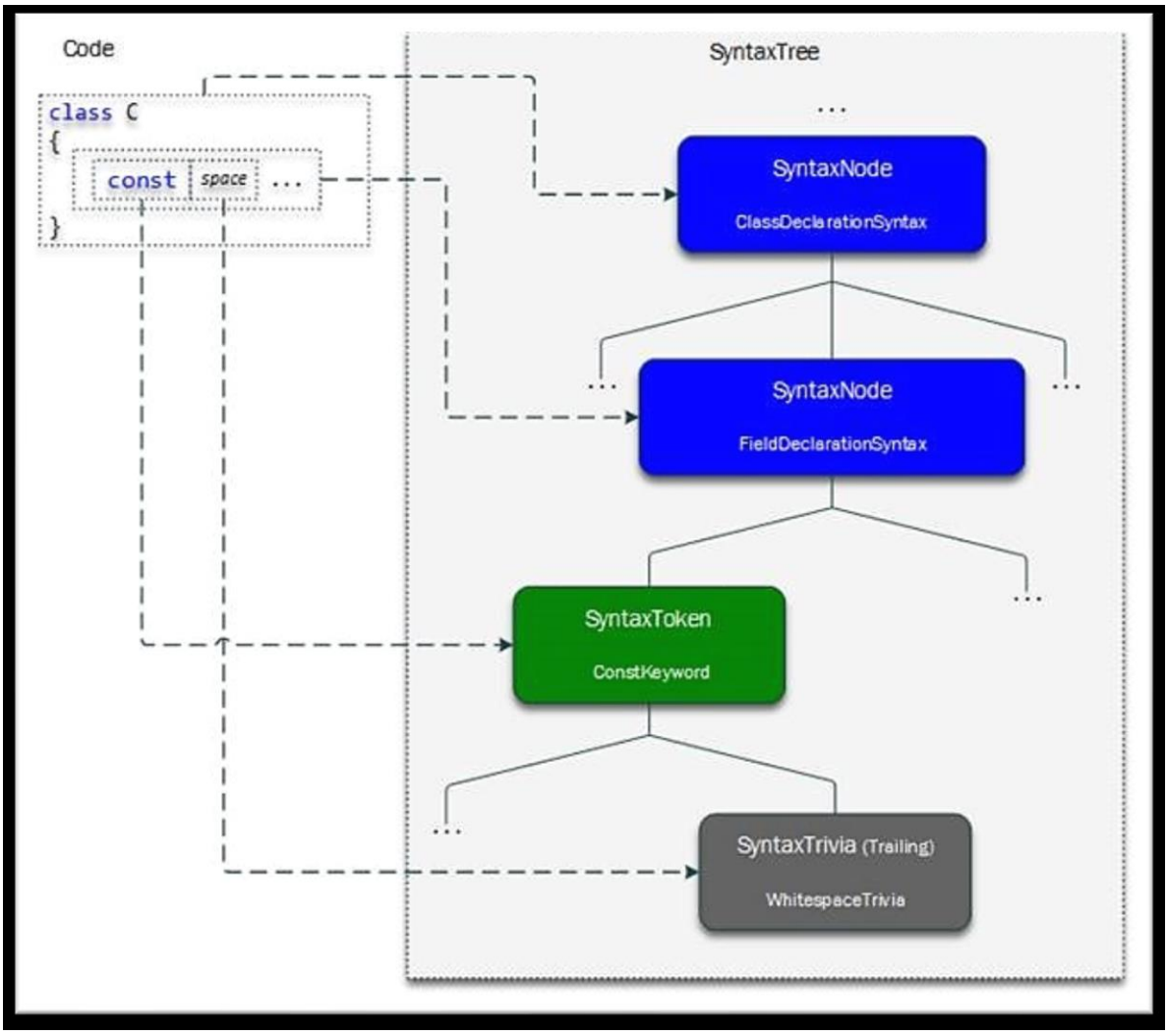

Рисунок 4.3 – Структура синтаксичного дерева

Синтаксичні токени є терміналами граматики мови, що представляють найменші синтаксичні фрагменти коду. Вони ніколи не бувають предками інших вузлів або токенів. Синтаксичні маркери складаються з ключових слів, ідентифікаторів, литералов і розділових знаків.

Додаткова синтаксична інформація (syntax trivia) являє такі частини тексту в вихідному коді, як пробіли, коментарі та директиви препроцесора. Для опису всіх видів trivia використовується один тип Microsoft.CodeAnalysis.SyntaxTrivia.

Кожному вузлу, токену або елементу trivia відомо його місце в тексті вихідного коду і число символів, з яких він складається. Положення в тексті представлено у вигляді 32-розрядного цілого числа, що є відлічуваним від нуля індексом char. Об'єкт TextSpan позначає початкове положення і кількість символів, представлені у вигляді

цілих чисел. Якщо TextSpan має нульову довжину, він позначає розташування між двома символами.

У кожного вузла є дві властивості TextSpan: Span і FullSpan. Властивість Span є текстовим діапазоном від початку першого токена в піддереві вузла до кінця останнього токена. Цей діапазон не включає в себе ніякі початкові або кінцеві елементи trivia. Властивість FullSpan є текстовим діапазоном, що включає в себе звичайний діапазон вузла, а також діапазон будь-яких початкових або кінцевих елементів trivia.

4.3 Опис реалізації рефакторинг провайдерів

Кожен клас який представляє рефакторінг провайдер повинен успадковуватися від абстрактного класу CodeRefactoringProvider та бути позначений атрибутом ExportCodeRefactoringProvider, в інакшому IDE не зможе знайти створений провайдер (рисунок 4.4).

```
1   using System.Linq;
2   using Microsoft.CodeAnalysis.CodeRefactorings;
3   using Microsoft.CodeAnalysis;
4   using System.Threading.Tasks;
5   using Microsoft.CodeAnalysis.CSharp.Syntax;
6   using Microsoft.CodeAnalysis.CSharp;
7   using Microsoft.CodeAnalysis.Formatting;
8   using RefactoringCSharp.Extension;
9   using RefactoringCSharp.Infrastructure;
10
11  namespace RefactoringCSharpn.Provider
12  {
13      [ExportCodeRefactoringProvider(LanguageNames.CSharp, Name = nameof(AddNullCheckProvider))]
14      public class AddNullCheckProvider : CodeRefactoringProvider
15      {
16          public override async Task ComputeRefactoringsAsync(CodeRefactoringContext context)...
```

Рисунок 4.4 – Оголошення рефакторинг провайдера

Код по реалізації рефакторінгу знаходиться в методі ComputeRefactoringAsync що приймає вхідний пареметер типу CodeRefactoringContext. Даний тип містить

інформацію про поточний документ (Document), поточне місце в тексті (TextSpan), токен для скасування операції (CancellationToken).

Клас CodeRefactoringContext надає можливість зареєструвати рефакторінг дію в Quick Actions меню. Окрім того вище описаний метод є асинхронним щоб не сповільнювати роботу IDE, оскільки будь які дії по аналізу коду, отриманню метаданих по елементам документу може забирати певний час.

Розглянемо реалізацію рефакторинга, який додає умову на рівність об’єкта та значення null. Блок-схема даного рефакторинга представлена на рисунку 4.5. При запуску Visual Studio відбувається виконання методів в асинхронному режимі для класів які мають предка CodeRefactoringProvider.

Логіка методу ComputeRefactoringAsync обчислює вузол у дереві синтаксису, на якому потрібно надати рефакторинг. Дивлячись на рефакторинг, який додає умову на рівність null, він починається з пошуку вузла з SyntaxTree. Далі через властивість Span з контексту отримаємо поточне місце в коді.

Для поточного завдання нам слід перевірити що елемент в якому стоїть курсор є ідентифікатором, оскільки ми шукаємо об’єкт або змінну типу значення Nullable типу. Якщо ж знайдений елемент не ідентифікатор, то ми закінчуємо роботу методу.

Наступний крок це перевірка зовнішнього елемента навколо поточного оператору з ідентифікатором. Розглядаємо випадок в якому активний оператор може знаходитись в блоці умовного оператора, тоді переконуємось, що умова не має перевірки яку ми плануємо додати.

Далі іде етап створення оператора умови з перевіркою, що ідентифікатор не дорівнює null. В тіло оператору ‘if’ поміщається оператор який використовує ідентифікатор. Після створення ми замінюємо оператор який отримали на почтаковому етапі на згенерований оператор умови.

Після пройдених етапів, реєструється рефакторинг за допомогою методу RegisterRefactoring в класі CodeRefactoringContext. Він реєструє делегат, який повертає значення типу Task<Solution>, яке оновлює рішення за допомогою коду рефакторингу.

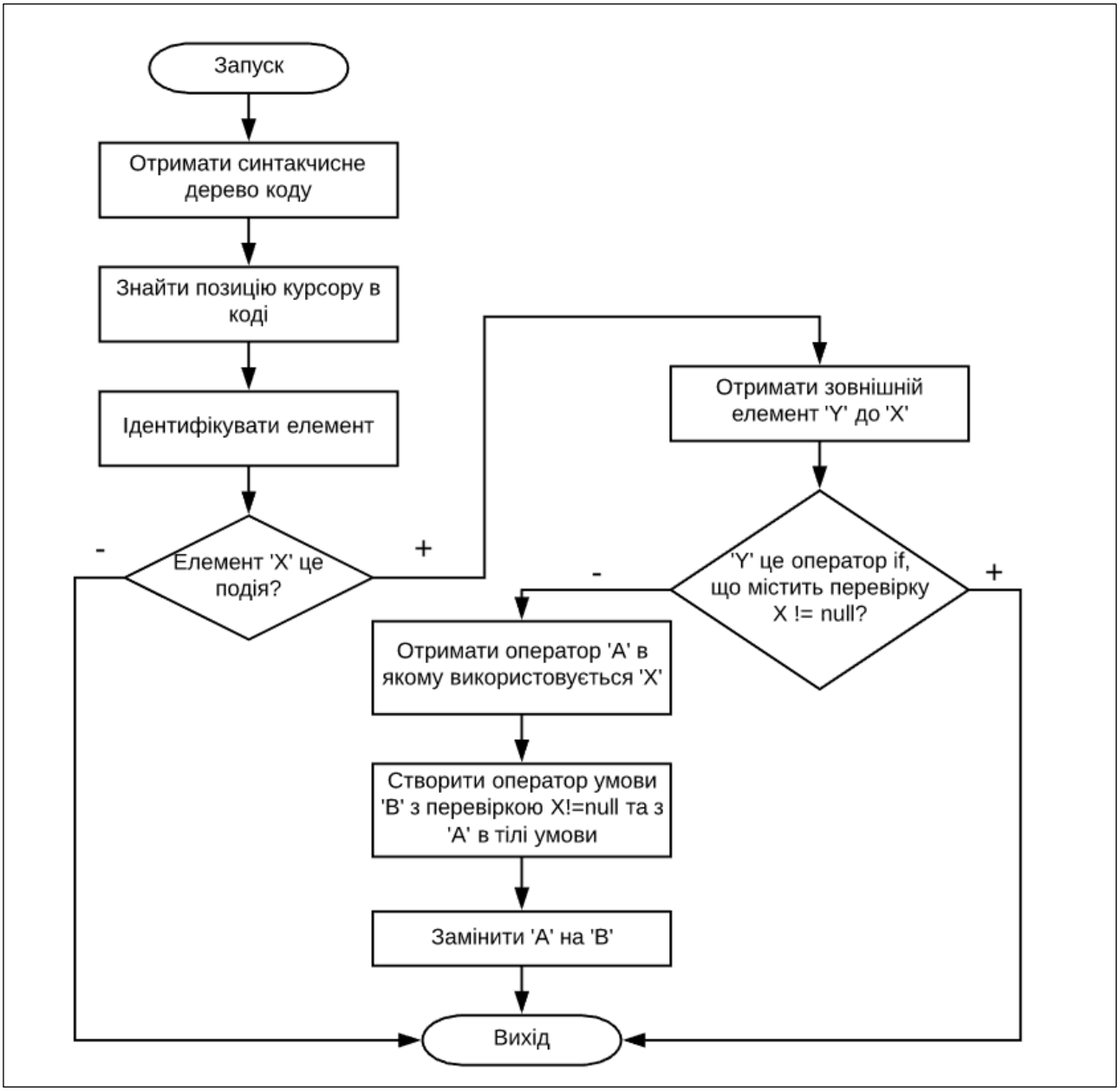

Рисунок 4.5 – Блок-схема рефакторингу “Add Null Check”

Наведена блок-схема (рисунок 4.6) демонструє логіку створення нового пункта Quick Actions меню “Create changed event”. Як і для коду кожного рефакторингу визначається синтаксичне дерево коду. На наступному етапі знаходимо активну позицію в коду та починаємо ідентифікацію даного елементу. Пропозиція даного рефакторингу буде відображатись тільки для типу подія, яка має валідний тип, тобто делегат в якого визначений метод Invoke().

До фінального етапу відноситься генерація методу виклику події. На ньому

визначається за допомогою доступних API, ща надає платформа Roslyn, сигнатура делегату. Дана сигнатура використовується в методі виклику події, що створюється.

Після генерації іде процес заміни існуючого синтаксичного дерева, через додавання перед оголошенням події створеного методу.

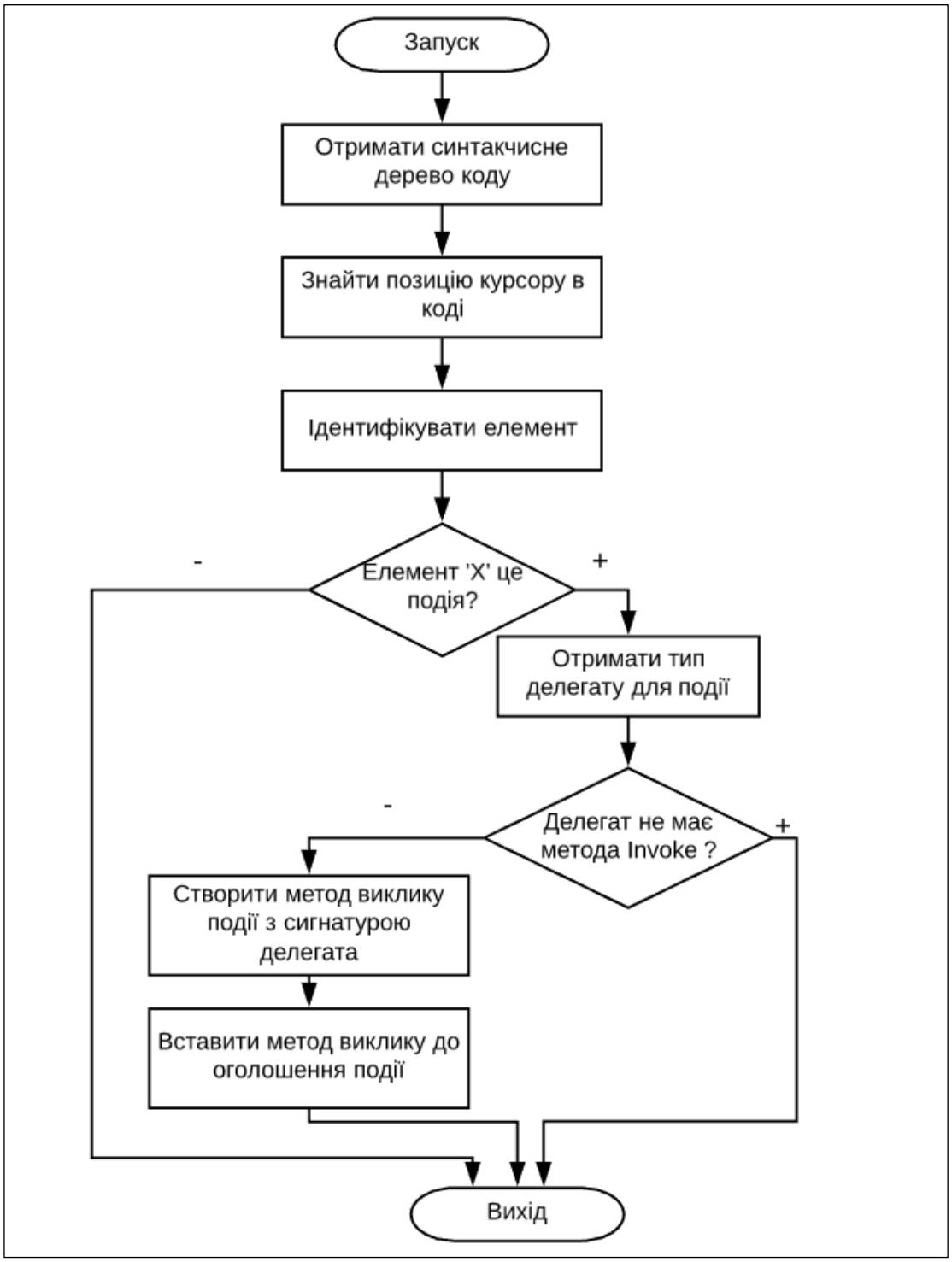

Рисунок 4.6 – Блок-схема рефакторингу “Create event invocator”

4.4 Опис рефакторинг функціоналу

Програмний продукт налічує 26 рефакторинг методів (таблиця 4.1), які виконують як генерацію нового коду так і оптимізацію існуючого.

№	Назва рефакторингу
1	Add null check
2	Extract anonymous method
3	Create changed event
4	Create delegate
5	Create event invocator
6	Create missing switch labels
7	Generate switch labels
8	Initialize readonly auto-property from parameter
9	Iterate via 'foreach'
10	To lambda expression
11	Replace 'if' with 'return'
12	Convert 'if' to 'switch' operator
13	To conditional ternary expression
14	Replace with '??' operator
15	Import static class with using
16	To property with backing field
17	Convert cast to 'as'
18	To 'var'
19	Split declaration and assignment
20	Add another accessor
21	Use if ('x'.TryGetValue(out var 'y'))

№	Назва рефакторингу
22	To expression body
23	Add null check for parameter
24	To interpolated string
25	Convert to auto-property
26	Use 'as' and check for null

Таблиця 4.1 – Рефакторинг методи

Програмне розширення надає рефакторинг, який можна класифікувати на три категорії (рисунок 4.7):

— генерація нового коду;

— усунення помилок, які можуть виникнути на етапі виконання;

— трансформація коду до сучаної форми запису.

Рисунок 4.7 – Класифікація функціоналу компоненту

До першої категорії відноситься створення події на зміну властивості (рисунок 4.8). Часто виникає ситуація в якій потрібно створити подію, що буде реагувати на

зміну властивості об’єкта. Дані події зазвичай мають однаковий синтаксис тому було прийнято рішення автоматизувати створення такої події. Користувачу достатньо поставити курсор на назві властивості для якої необхідно згенерувати подію типу EventHandler, тобто метод який буде обробляти дану подію повинен приймати два вхідних аргументи типу Object та EventArgs та нічого не повертати, тобто мати тип вихідного значення void. Разом з методом створюється метод який безпечно викликає подію. Перед викликом події перевіряється що для неї було зареєстровано обробник події.

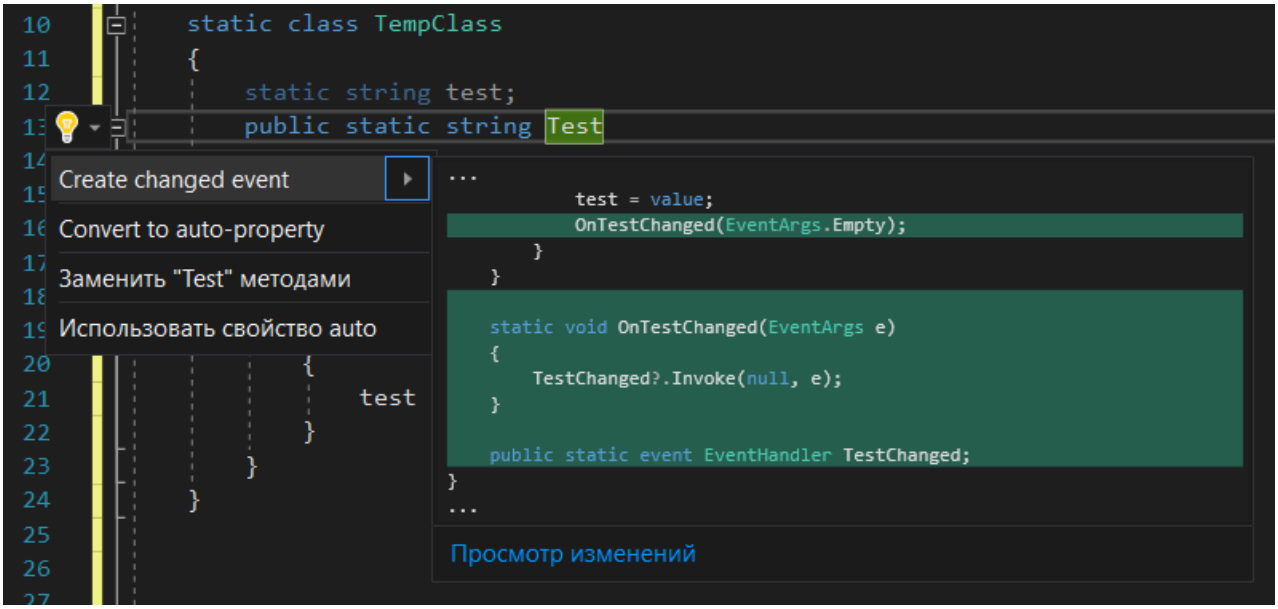

Рисунок 4.8 – Рефакторинг “Create changed event”

Автоматизовано створення методу безпечного виклику події (рисунок 4.9). Користувачу достатньо створити подію та скористатись запропонованою опцією.

При створення події з типом делегату який ще не оголошено є можливість автоматичного створення делегату з стандартною сигнатурою для події, а саме з значенням що повертається void та двома вхідними параметрами: власник типу object та аргументи виклику типу EventArgs.

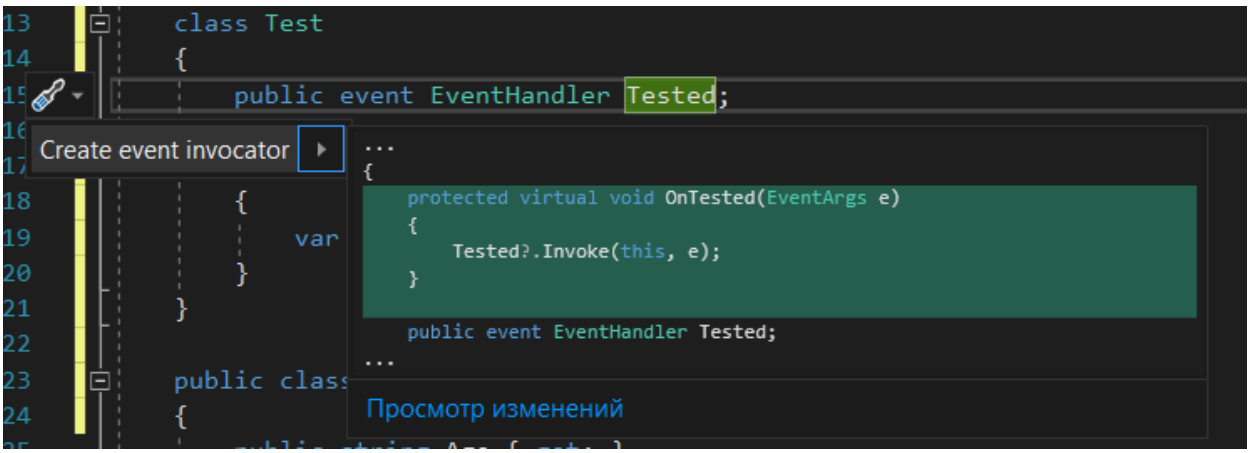

Рисунок 4.9 – Рефакторинг “Create event invocator”

При написанні оператору switch розробник може скористатись генерацією всіх можливих варіантів для значення що порівнюється. Такий рефакторинг значно пришвидшує написання коду особливо для перерахувань, для прикладу таких як статуси виконанння HTTP запитів.

Ініціалізація властивостей та полів об’єкту класу через конструктор представлена рефакторингом (рисунок 4.9), який дозволяє написати вхідний параметр для констурктора та згенеревати властивість тільки для читання. Що дозволяє без зайвих зусиль створювати та одразу ініціалізувати властивості класу відразу на етапі розробки.

Для типів які реалізують інтерфейс IEnumerable є можливість генерації оператору foreach який дозволяє циклічно звертатись до кожного елементу колекції окремо.

Наведена блок-схема (рисунок 4.8) демонструє логіку створення нового пункта Quick Actions меню “Create changed event”. Як і для коду кожного рефакторингу визначається синтаксичне дерево коду. На наступному етапі знаходимо активну позицію в коду та починаємо ідентифікацію даного елементу. Пропозиція даного рефакторингу буде відображатись тільки для типу подія, яка має валідний тип, тобто делегат в якого визначений метод Invoke().

До фінального етапу відноситься генерація методу виклику події. На ньому

визначається за допомогою доступних API, ща надає платформа Roslyn, сигнатура делегату. Дана сигнатура використовується в методі виклику події, що створюється.

Після генерації іде процес заміни існуючого синтаксичного дерева, через додавання перед оголошенням події створеного методу.

При написанні оператору switch розробник може скористатись генерацією всіх можливих варіантів для значення що порівнюється. Такий рефакторинг значно пришвидшує написання коду особливо для перерахувань, для прикладу таких як статуси виконанння HTTP запитів.

Ініціалізація властивостей та полів об’єкту класу через конструктор представлена рефакторингом (рисунок 4.10), який дозволяє написати вхідний параметр для констурктора та згенеревати властивість тільки для читання. Що дозволяє без зайвих зусиль створювати та одразу ініціалізувати властивості класу відразу на етапі розробки.

Для типів які реалізують інтерфейс IEnumerable є можливість генерації оператору foreach який дозволяє циклічно звертатись до кожного елементу колекції окремо.

Рисунок 4.10 – Рефакторинг “Initialize readonly auto-property from parameter”

Друга категорія дозволяє примінити рефакторинг до коду, який регулярно приводить до помилок на етапі виконання. При розробці часто забувають про те що типи посилань мають значення за замовчуванням null, тому на екземплярах типу викликають методи або отримують значення властивостей не перевіряючи чи об’єкт

не дорівнює null, що призводить до неочікуваного виключення NullReferenceException. Щоб уникнути подібних ситуацій, запропоновано рефакторинг (рисунок 4.11) який додає таку перевірку до вхідних параметрів методу та змінних в середині метода, які належать до типу посилань або до Nullable типу.

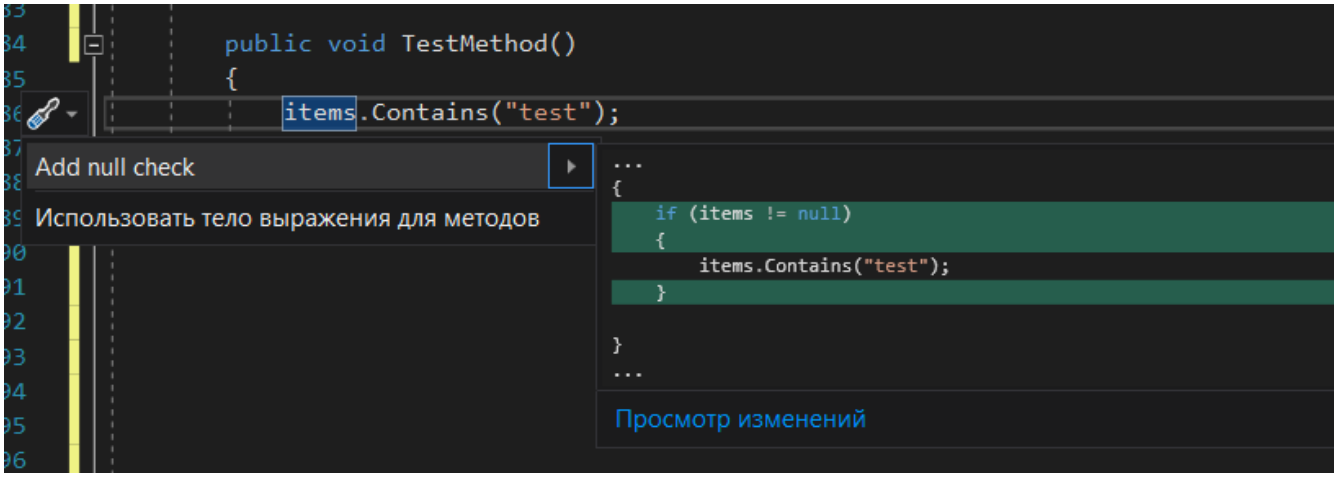

Рисунок 4.11 – Рефакторинг “Add null check”

Для типу Dictionary доступ до значення по ключу відбувається через індексатор. Недоліком є те що індексатор приймає ключ як параметр, але якщо вказаний ключ не існує, тоді виникає KeyNotFoundException. Для того щоб забезпечити коректне виконання програми, в випадку коли ми не знаємо чи заданий ключ існує, рефакторинг замінити індексатор на метод TryGetValue (рисунок 4.12). Метод TryGetValue повертає false, якщо не вдалося знайти за певним ключем значення.

Наступний рефакторинг (рисунок 4.13) забезпечує безпечне приведення змінної з похідного типу до базового типу, за допомогою оператора as. Безпека означає, що на етапі виконання в коді не виникне виключення, а змінна, якій буде привласнено значення, отримає значення null, якщо приведення не зможе бути виконано. Якщо типи є сумісними, використання безпечного приведення завжди буде успішним.

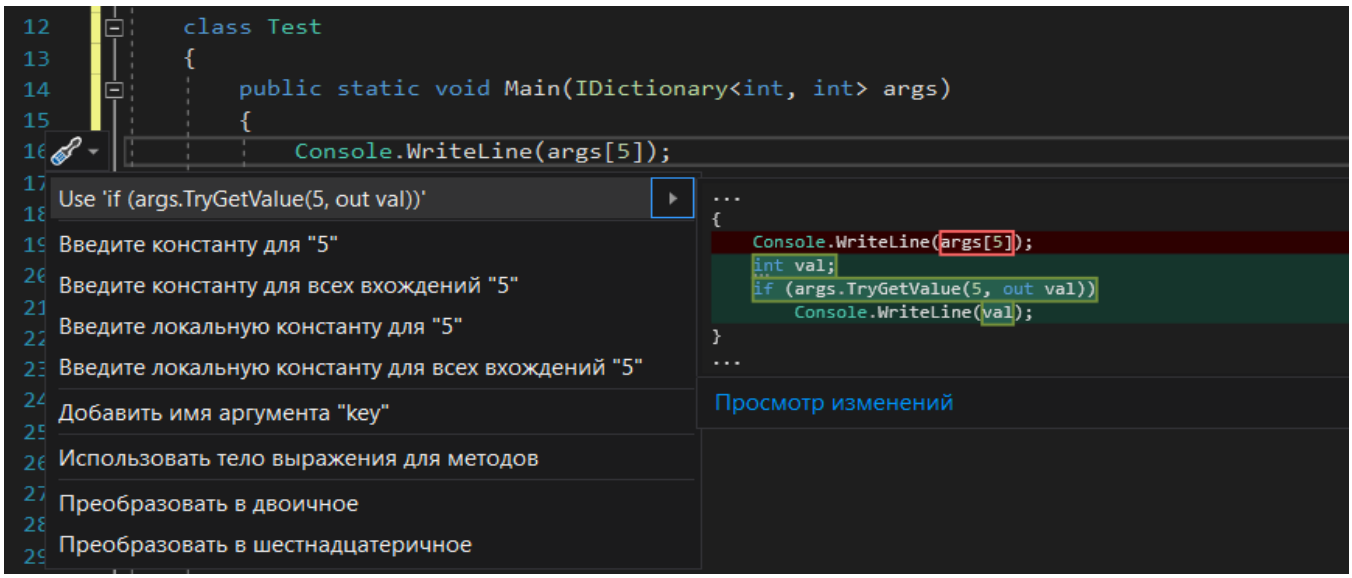

Рисунок 4.12 – Рефакторинг “Convert to method TryGetValue”

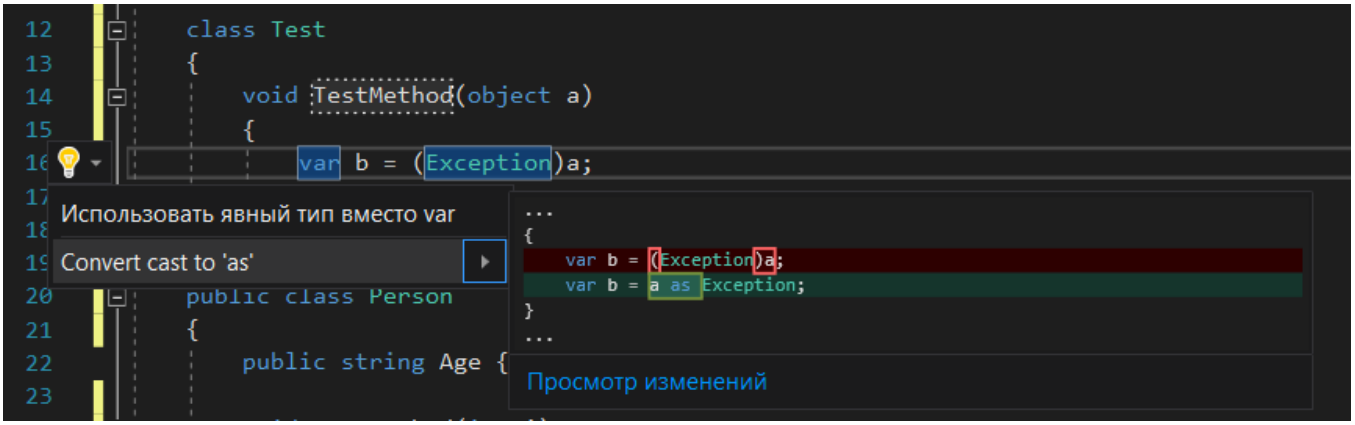

Рисунок 4.13 – Рефакторинг “Convert cast to ‘as’”

Третя категорія надає функціонал рефакторингу для трансформації поточного коду до сучасної форми запису. Мова програмування C# надає багатий синтаксис, який дозволяє створювати альтернативні варіанти коду, які в свою чергу значно спрощують код та збільшують читабельність коду. Основними засобами для втілення даних цілей є оператори умови ‘?:’, ‘??’, інтерполяція та інші.

Розглянемо детальніше оптимізацію оператора умови. Рефакторинг перетворює блоки ‘if-else’ умовного оператору у вираз, який використовує тернарний оператор ‘?:’. Даний вид рефакторингу зменшує кількість коду.

Рефакторинг оптимізації до тернарного оператора доступний, коли курсор знаходиться на операторі ‘if’, що має відповідний блок ‘else’. Блоки ‘if’ і ‘else’

повинні містити одну операцію. Ці блоки повинні мати аналогічні оператори ‘return’ (рисунок 4.14), або оператори присвоєння (рисунок 4.15).

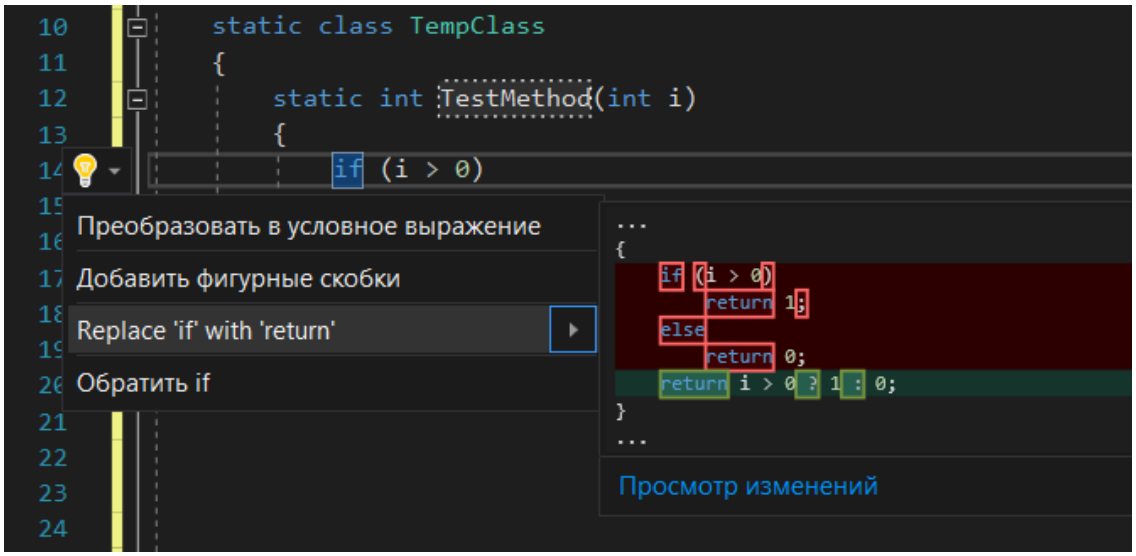

Рисунок 4.14 – Рефакторинг “Replace ‘if’ with ‘return’”

Рефакторинг який конвертує метод String.Format() до синтаксису інтерполяції (рисунок 4.16). Інтерполяція представлена спеціальним знаком ‘$’, який ідентифікує string літерал як інтерпольований рядок. Інтерпольований рядок - це рядковий літерал, який може містити інтерпольовані вираження.

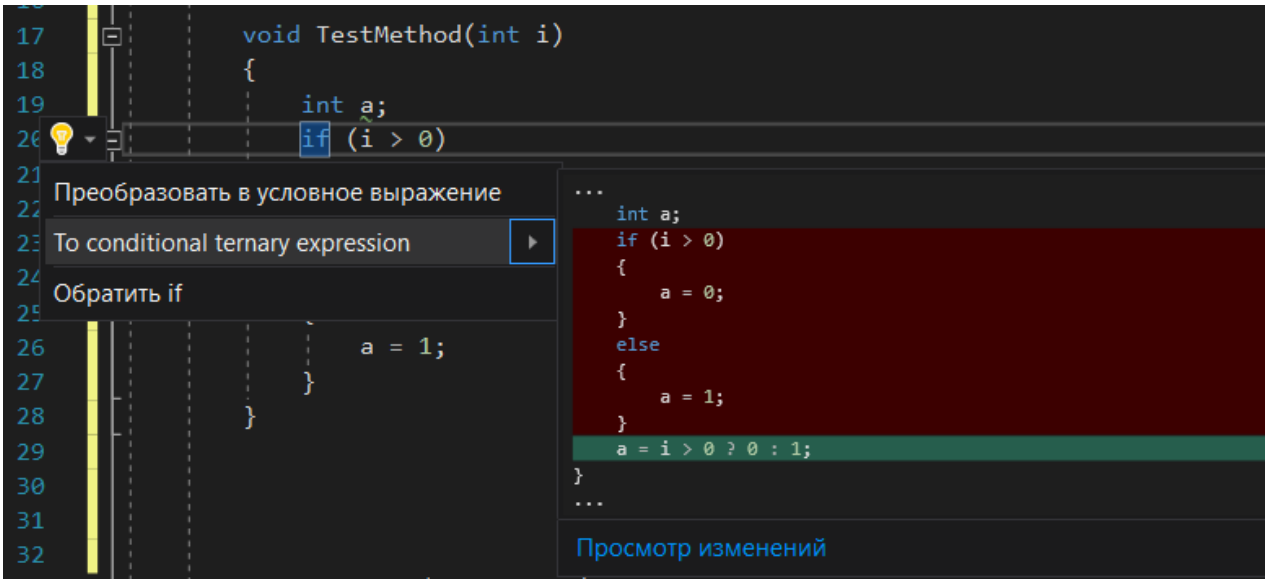

Рисунок 4.15 – Рефакторинг “To conditional ternary expression”

При конвертації інтерпольованого рядка в результуючий елемент виникають інтерпольовані вирази заміни рядкових елементів на результати виразів. Ця можливість доступна в C # 6 і більш пізніх версіях.

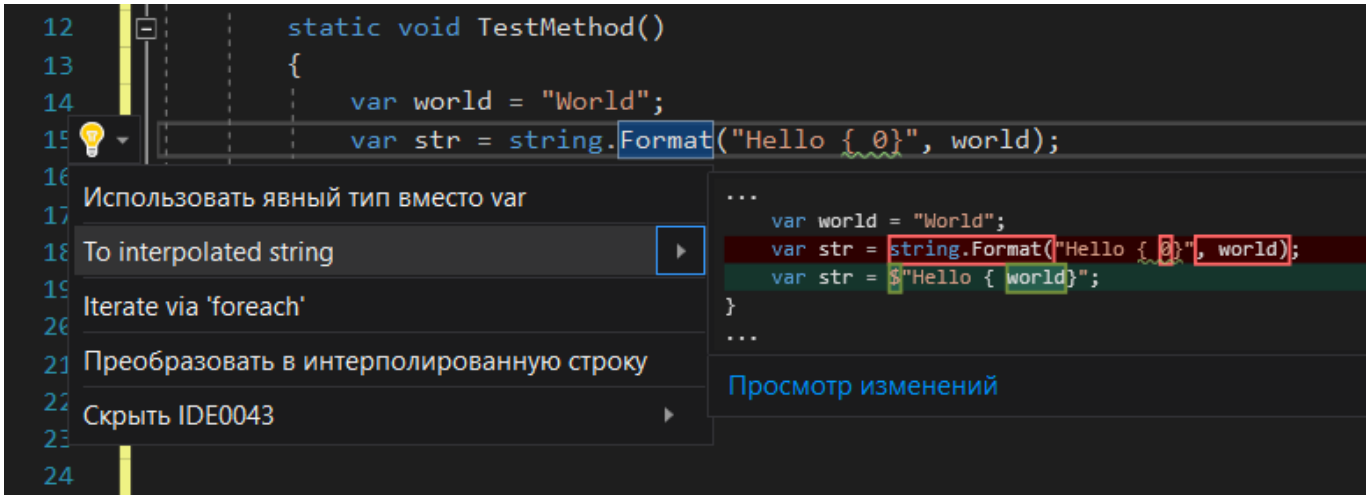

Рисунок 4.16 – Рефакторинг “To interpolated string”

Інтерполяція рядків надає більш зрозумілий і зручний синтаксис для створення форматованих рядків по відношенню до функціонального складу форматування рядків.

Замість того, щоб передаватити анонімний метод делегату або використати лямбда, можна використовувати лямбда-вираз. Це справедливо лише для методів, що містять одну операцію.

Рефакторинг “To lambda expression” надає користувача зручний спосіб конвертації ананімного методу до лямбда-виразу.

Для створення анонімних функцій можна використовувати як лямбда-вирази, так і анонімні методи, проте лямбда-вирази забезпечують більш оптимізований синтаксис.

У наведеному прикладі (рисунок 4.17) компонент рефакторингу пропонує призначити лямбда-вираз делегату типу Action<int, int>, а не використовувати не читабельний анонімний метод:

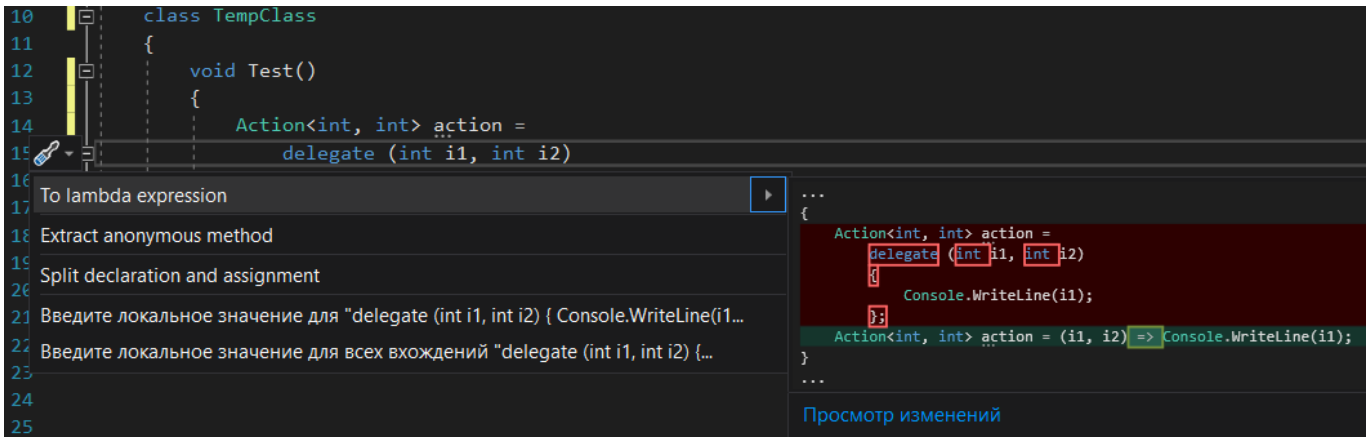

Рисунок 4.17 – Рефакторинг “To lambda expression”

Рефакторинг “Replace type with ‘var’” (рисунок 4.18) дозволяє в зручний спосіб замінити явний тип на тип “var”. Локальні змінні можуть бути оголошені без надання явного типу. Ключове слово var вказує компілятору вивести тип змінної з виразу на правій стороні оператора ініціалізації. Визначений тип може бути вбудованим типом, анонімним типом, користувацьким типом або типом, визначеним у бібліотеці класів .NET Framework. Перевагою такого використання є те, що читабельність коду підвищується особливо помітно в випадках комплексних типів.

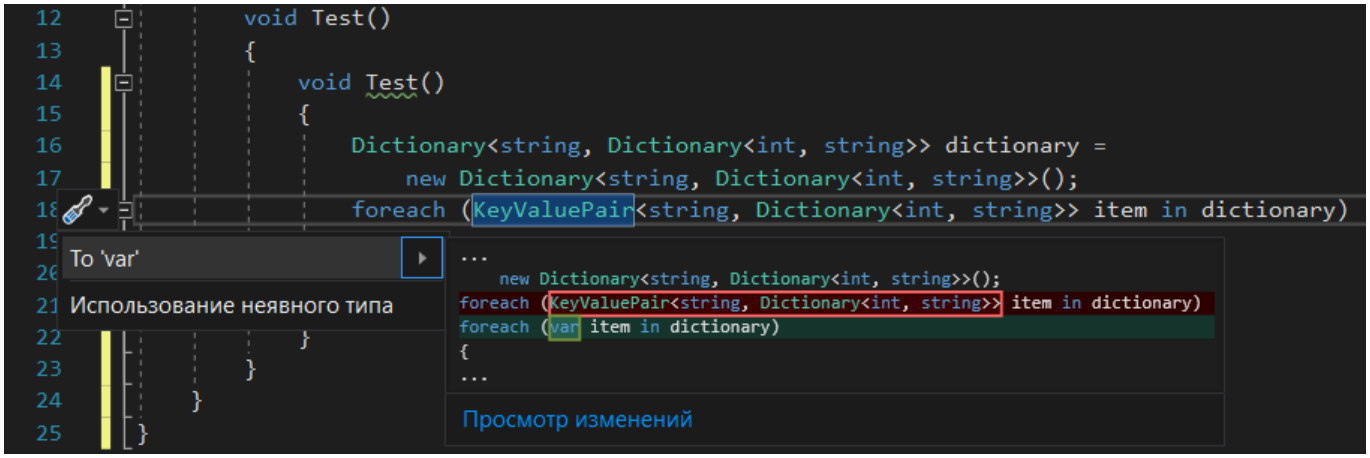

Рисунок 4.18 – Рефакторинг “Replace type with ‘var’”

Висновки до розділу

Було розроблено 26 рефакторинг методів, що дозволяють.

Програмне розширення надає рефакторинг, який підвищує продуктивність розробки, шляхом генерація нового коду, усунення помилок, які можуть виникнути на етапі виконання, трансформація коду до сучаної форми запису.

Рефакторинг методи інтегруються в інтерфейс IDE Visual Studio, що дозволяє користувачу бачити доступні рефакторинги для даного блоку коду.

5 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Для встановлення компоненту рефакторінгу користувач повинен мати встановлену одну з наступних IDE Microsoft.VisualStudio.Community, Microsoft.VisualStudio.Pro або Microsoft.VisualStudio.Enterprise версії 15.0 і вище.

Для встановлення розширення та керуванням ним користувач може використовувати діалогове вікно “Керувати розширеннями”. Щоб відкрити діалогове вікно, виберіть “Розширення” > “Керувати розширеннями”. Далі слід знайти в полі пошуку RefactoringCSharp розширення та натиснути кнопку “Встановити”.

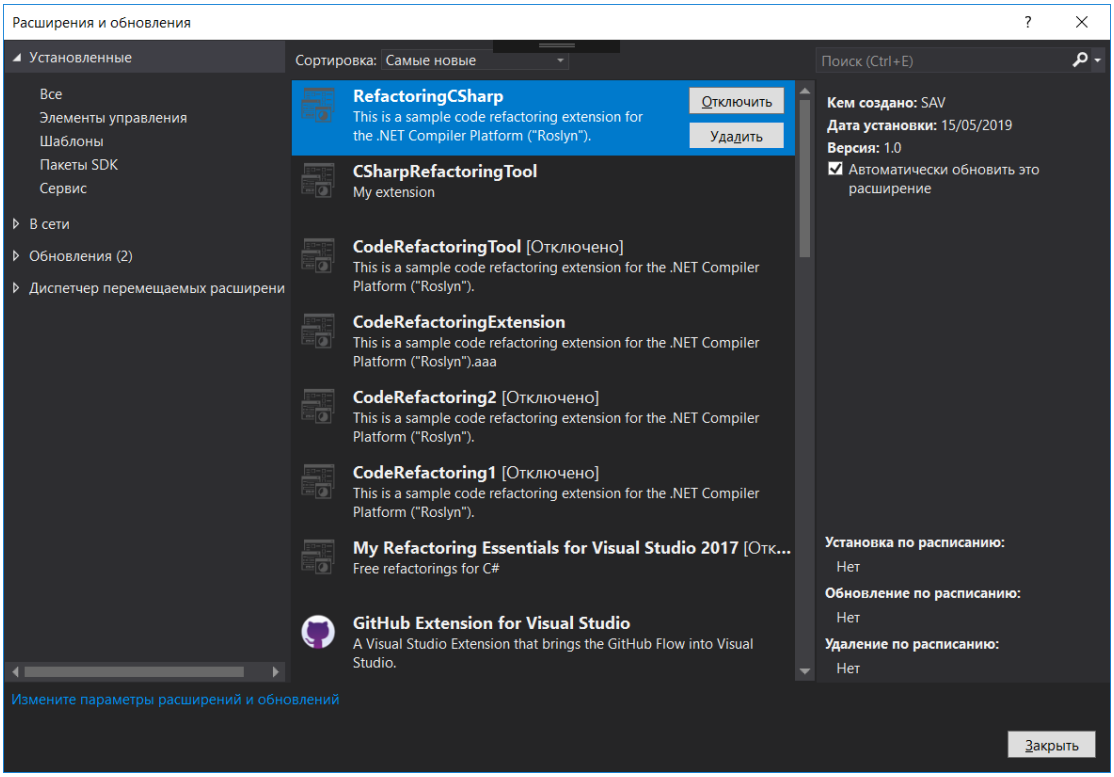

Рисунок 5.1 – Вікно “Розширення та оновлення”

Буде надано можливість встановлення розширення через інсталятор. Натиснувши двічі на інсталятор (файл з розширенням .vsix) і натиснувши клавішу

Enter або кнопку “Встановити”. Після цього дотримуйтесь інструкцій. Після установки таке розширення можна ввімкнути, вимкнути або видалити в діалоговому вікні “Керувати розширеннями”.

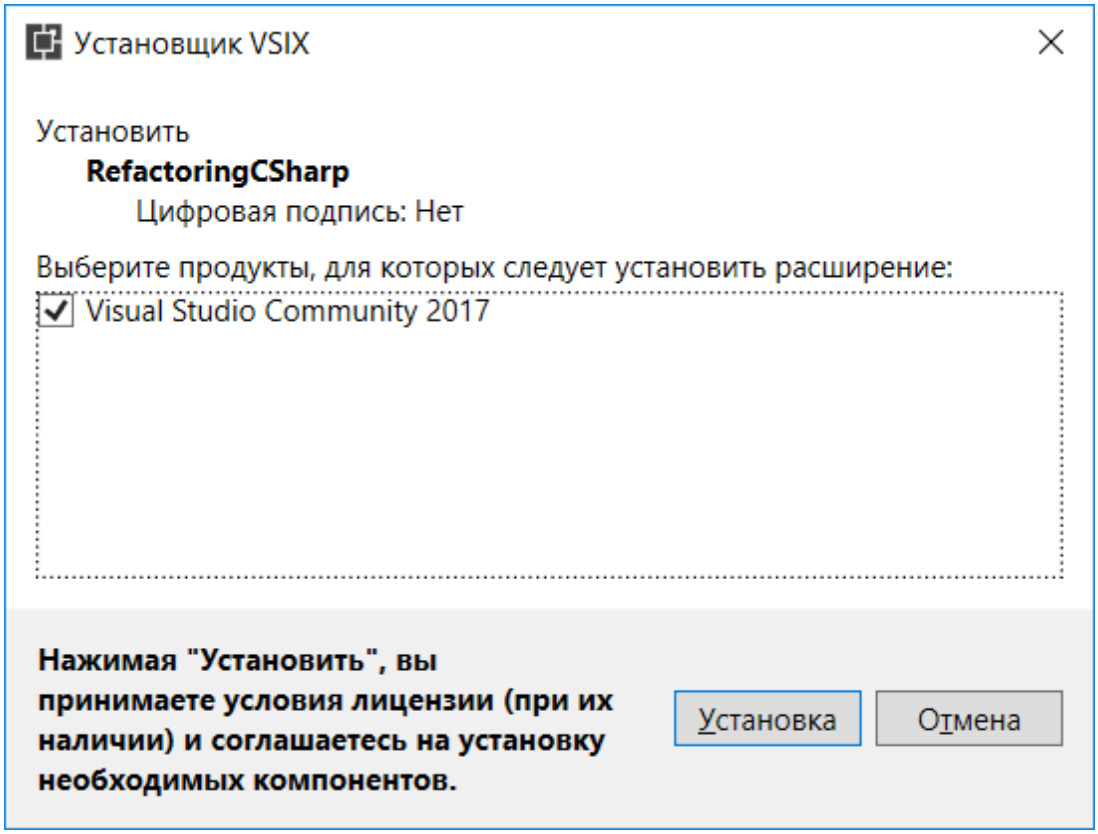

Рисунок 5.2 – Вікно інсталятора

Для користування компонентом рефакторінгу потрібно поставити курсор на фрагмент коду який користувач хоче покращити. Після цього натиснути Alt+Enter для того щоб відобразилися доступні рефакорінг провайдери та з Quick Actions меню вибрати рефакторінг який буде примінено до даного фрагменту коду.

Висновки до розділу

Компонент автоматизованого рефакторингу потребує від користувача знань та можливостей середовища розробки Visual Studio.

Компонент рефакторингу в IDE Visual Studio можу бути встановлений через зручний для користувача спосіб (файл інсталяції або вікно Visual Studio “Розширення та оновлення”). Для користувача надано інструкції для користування компонентом автоматизованого рефакторингу.

Висновки

У ході аналізу існуючого програмного забезпечення для автоматичного рефакторінгу, було проаналізовано недоліки існуючих рішень. Аналіз показав, що існуючі рефакторинг компоненти інтегровані в IDE Visual Studio вирішують окрім рефакторингу інші завдання, такі як підтримка юніт тестування, створення синтаксичного аналізатора, додавання нових функцій до редактору, цим самим маючи негативний вплив на продуктивність роботи IDE.

На основі отриманого аналізу було прийнято рішення розробити новий компонент для розширення функціоналу рефакторингу в IDE Visual Studio. Завдяки використанню API, що надає платформа компіляторів .Net, вплив на продуктивність IDE є мінімальною. Новий компонент рефакторингу в IDE Visual Studio є відкритим до загального користування продуктом.

Розроблений програмний продукт дозволяє автоматизувати розпізнавання фрагментів коду для яких є можливим рефакторинг та сам процес рефакторингу шляхом використання додаткових опцій в Quick Actions меню.

Проведено огляд методів і засобів розробки програмної системи. Обґрунтовано вибір створення програмної системи, заснованої на SDK та .Net Compiler Platform.

За результатами виконання тестових завдань підтверджена коректність виконання програми, отже система відповідає поставленим вимогам.

Користувачами системи можуть бути розробники, що використовують IDE Visual Studio та потребують зручного та зрозумілого автоматизованого рефакторинг інструменту для мови C#. Програмне забезпечення може бути використано на будь-якій операційній системі, на якій встановлено IDE Visual Studio.

Компонент рефакторингу в IDE Visual Studio можу бути встановлений через зручний для користувача спосіб (файл інсталяції або вікно Visual Studio “Розширення та оновлення”). Для користувача надано інструкції для користування компонентом автоматизованого рефакторингу.

Список використаних джерел

1. Джон Рихтер. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#: книга / Джон Рихтер. – 4-е изд, 2017. – 896с.

2. Роберт К. Мартин. Чистый код. Создание, анализ и рефакторинг: книга / Роберт К. Мартин. Питер, 2016. – 464с.

3. Мартин Фаулер. Рефакторинг. Улучшение существующего кода: книга / Мартин Фаулер. Вильямс, 2016. – 448с.

4. Стив Макконнелл. Совершенный код: книга / Мартин Фаулер. БХВ-Петербург, 2016. – 896с.

5. Стив Макконнелл. Профессиональная разработка программного обеспечения: книга / Стив Макконнелл. Символ-Плюс, 2007. – 240с.

6. Manish Vasani. Roslyn Cookbook: book / Manish Vasani. Packt Publishing, 2017. – 350 pages.

7. Nick Harrison. Code Generation with Roslyn: book / Nick Harrison. Apress, 2017. – 105 pages.

8. Joshua Kerievsky. Refactoring to Patterns: book / Joshua Kerievsky. Addison-Wesley Professional. – 1 edition, 2004. 400 pages.

9. Ed Crookshanks. Just Enough Debugging: Debugging, Refactoring, and Unit Tests: book / Ed Crookshanks. CreateSpace Independent Publishing Platform. 2018. – 110 pages.

10. Peter Ritchie. Refactoring with Microsoft Visual Studio 2010: book / Peter Ritchie. Packt Publishing. 2010 – 372 pages.

11. Danijel Arsenovski. Professional Refactoring in C# & ASP.NET: book / Danijel Arsenovski. Wrox; – 1 edition, 2009 – 517 pages.

12. Girish Suryanarayana. Refactoring for Software Design Smells: Managing Technical Debt: book / Girish Suryanarayana. Morgan Kaufmann; 1 edition, 2014 – 258 pages.

13. Genevieve Warren. Get started with Roslyn analyzers / Genevieve Warren // MSDN, (https://docs.microsoft.com/en-us/visualstudio/extensibility/getting-started-with-roslyn-analyzers?view=vs-2019).
14. Joey Robichaud. Roslyn Overview / Joey Robichaud // GitHub, (https://github.com/dotnet/roslyn/wiki/Roslyn-Overview?fbclid=IwAR3v_xiJtuA0vb5lSmt_KRFfefDI4nOKr0TX4K7s-zUyZaRCbUju3C_OKZA).

ДОДАТОК А

Компонент рефакторингу в інтегрованому середовищі розробки Visual Studio

Специфікація

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТР5166_19Б

Аркушів 1

Київ 2019

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ”_ ТЕФ_АПЕПС_ ТР5166_19Б	Записка.docx	Пояснювальна записка
Компоненти		
УКР.НТУУ”КПІ”_ ТЕФ_АПЕПС_ТР5 166_19Б 12-1	RefactoringCSharp.vsix RefactoringCSharp.dll	Основні компоненти.
УКР.НТУУ”КПІ”_ ТЕФ_АПЕПС_ ТР5166_19Б 13-1	Додаток В.doc	Опис програмного модуля

ДОДАТОК Б

Компонент рефакторингу в інтегрованому середовищі розробки Visual Studio

Текст програми

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТР5166_19Б 12-1

Аркушів 8

Київ 2019

```
using System.Linq;
using System.Threading;
using System.Collections.Generic;
using Microsoft.CodeAnalysis.CodeRefactorings;
using Microsoft.CodeAnalysis;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.Text;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.Simplification;
using Microsoft.CodeAnalysis.Formatting;
using RefactoringCSharp.Provider;
using RefactoringCSharp.Infrastructure;

namespace RefactoringEssentials.CSharp.CodeRefactorings
{
    [ExportCodeRefactoringProvider(LanguageNames.CSharp, Name =
nameof(CheckIfParameterIsNullCodeRefactoringProvider))]
    public class CheckIfParameterIsNullCodeRefactoringProvider :
SpecializedCodeRefactoringProvider<ParameterSyntax>
    {
        protected override IEnumerable<CodeAction> GetActions(Document
document, SemanticModel semanticModel, SyntaxNode root, TextSpan span,
ParameterSyntax node, CancellationToken cancellationToken)
        {
            if (!node.Identifier.Span.Contains(span))
                return Enumerable.Empty<CodeAction>();
            var parameter = node;
            var bodyStatement =
parameter.Parent.Parent.ChildNodes().OfType<BlockSyntax>().FirstOrDefault();
            if (bodyStatement == null)
                return Enumerable.Empty<CodeAction>();

            var parameterSymbol = semanticModel.GetDeclaredSymbol(node);
            var type = parameterSymbol.Type;
            if (type == null || type.IsValueType ||
HasNullCheck(semanticModel, parameterSymbol, bodyStatement))
                return Enumerable.Empty<CodeAction>();
            return new[] { CodeActionFactory.Create(
                node.Identifier.Span,
                "Add null check for parameter",
                GenerateNewNode(document,root,node,bodyStatement)
            ) };
        }

        private static System.Func<CancellationToken, Task<Document>>
GenerateNewNode(Document document, SyntaxNode root, ParameterSyntax node,
BlockSyntax bodyStatement)
        {
            return t =>
```

```
            {
                ExpressionSyntax parameterExpr;
                var paramName = node.Identifier.ToString();
                var parseOptions = root.SyntaxTree.Options as
CSharpParseOptions;
                if (parseOptions != null && parseOptions.LanguageVersion <
LanguageVersion.CSharp6)
                {
                    parameterExpr =
SyntaxFactory.LiteralExpression(SyntaxKind.StringLiteralExpression,
SyntaxFactory.Literal(paramName));
                }
                else
                {
                    parameterExpr = SyntaxFactory.ParseExpression("nameof(" +
paramName + ")");
                }
                IfStatementSyntax ifStatement =
BuildIfStatement(parameterExpr, paramName);

                var newBody =
bodyStatement.WithStatements(SyntaxFactory.List<StatementSyntax>(new[] {
ifStatement.WithAdditionalAnnotations(Formatter.Annotation,
Simplifier.Annotation) }.Concat(bodyStatement.Statements)));
                var newRoot = root.ReplaceNode((SyntaxNode)bodyStatement,
newBody);
                return Task.FromResult(document.WithSyntaxRoot(newRoot));
            };
        }

        private static IfStatementSyntax BuildIfStatement(ExpressionSyntax
parameterExpr, string paramName)
        {
            return SyntaxFactory.IfStatement(
                SyntaxFactory.BinaryExpression(SyntaxKind.EqualsExpression,
SyntaxFactory.IdentifierName(paramName),
SyntaxFactory.LiteralExpression(SyntaxKind.NullLiteralExpression)),
                SyntaxFactory.ThrowStatement(
                    SyntaxFactory.ObjectCreationExpression(

SyntaxFactory.ParseTypeName("System.ArgumentNullException"),

SyntaxFactory.ArgumentList(SyntaxFactory.SeparatedList(new[] {
SyntaxFactory.Argument(parameterExpr) })), null)
                )
            );
        }

        static bool HasNullCheck(SemanticModel semanticModel,
IParameterSymbol parameterSymbol, BlockSyntax bodyStatement)
        {
```

```
            foreach (var ifStmt in
bodyStatement.DescendantNodes().OfType<IfStatementSyntax>())
            {
                var cond = ifStmt.Condition as BinaryExpressionSyntax;
                if (cond == null || !cond.IsKind(SyntaxKind.EqualsExpression)
&& !cond.IsKind(SyntaxKind.NotEqualsExpression))
                    continue;
                ExpressionSyntax checkParam;
                if (cond.Left.IsKind(SyntaxKind.NullLiteralExpression))
                {
                    checkParam = cond.Right;
                }
                else if (cond.Right.IsKind(SyntaxKind.NullLiteralExpression))
                {
                    checkParam = cond.Left;
                }
                else
                {
                    continue;
                }
                var stmt = ifStmt.Statement;
                while (stmt is BlockSyntax)
                    stmt = ((BlockSyntax)stmt).Statements.FirstOrDefault();
                if (!(stmt is ThrowStatementSyntax))
                    continue;

                var param = semanticModel.GetSymbolInfo(checkParam);
                if (param.Symbol == parameterSymbol)
                    return true;
            }
            return false;
        }
    }
}


using System.Linq;
using Microsoft.CodeAnalysis.CodeRefactorings;
using Microsoft.CodeAnalysis;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.CSharp;
using System.Collections.Generic;
using System.Collections.Immutable;
using RefactoringCSharp.Extension;
using RefactoringCSharp.Infrastructure;
using RefactoringCSharp.Action;

namespace RefactoringCSharp.Provider
{
```

```
    [ExportCodeRefactoringProvider(LanguageNames.CSharp, Name = "Create event
invocator")]
    public class CreateEventInvocatorProvider : CodeRefactoringProvider
    {
        public override async Task
ComputeRefactoringsAsync(CodeRefactoringContext context)
        {
            var document = context.Document;
            if (document.Project.Solution.Workspace.Kind ==
WorkspaceKind.MiscellaneousFiles)
                return;
            var span = context.Span;
            if (!span.IsEmpty)
                return;
            var cancellationToken = context.CancellationToken;
            if (cancellationToken.IsCancellationRequested)
                return;
            var model = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

            var root = await
model.SyntaxTree.GetRootAsync(cancellationToken).ConfigureAwait(false);
            var token = root.FindToken(span.Start);
            if (!token.IsKind(SyntaxKind.IdentifierToken))
                return;

            var node = token.Parent as VariableDeclaratorSyntax;
            if (node == null)
                return;
            var declaredSymbol = model.GetDeclaredSymbol(node,
cancellationToken);
            if (declaredSymbol == null ||
!declaredSymbol.IsSuchKind(SymbolKind.Event))
                return;
            var invokeMethod =
declaredSymbol.GetReturnType().GetDelegateInvokeMethod();
            if (invokeMethod == null)
                return;

            context.RegisterRefactoring(
                CodeActionFactory.CreateInsertion(
                    span, "Create event invocator",
                    t2 =>
                    {
                        SyntaxNode eventInvocator =
CreateEventInvocator(document, declaredSymbol);
                        return Task.FromResult(new InsertionResult(context,
eventInvocator, declaredSymbol.ContainingType,
declaredSymbol.Locations.First()));
                    }
                )
```

```
            );
        }

        public static MethodDeclarationSyntax CreateEventInvocator(Document
document, ISymbol eventMember, bool useExplictType = false)
        {
            var options = document.Project.ParseOptions as
CSharpParseOptions;

            if (options != null && options.LanguageVersion <
LanguageVersion.CSharp6)
                return
CreateOldEventInvocator(eventMember.ContainingType.Name,
eventMember.ContainingType.IsSealed, eventMember.IsStatic, eventMember.Name,
eventMember.GetReturnType().GetDelegateInvokeMethod(), useExplictType);

            return CreateEventInvocator(eventMember.ContainingType.Name,
eventMember.ContainingType.IsSealed, eventMember.IsStatic, eventMember.Name,
eventMember.GetReturnType().GetDelegateInvokeMethod(), useExplictType);
        }

        public static MethodDeclarationSyntax CreateEventInvocator(Document
document, string declaringTypeName, bool isSealed, bool isStatic, string
eventName, IMethodSymbol invokeMethod, bool useExplictType = false)
        {
            var options = document.Project.ParseOptions as
CSharpParseOptions;

            if (options != null && options.LanguageVersion <
LanguageVersion.CSharp6)
                return CreateOldEventInvocator(declaringTypeName, isSealed,
isStatic, eventName, invokeMethod, useExplictType);
            return CreateEventInvocator(declaringTypeName, isSealed,
isStatic, eventName, invokeMethod, useExplictType);
        }

        static MethodDeclarationSyntax CreateMethodStub(bool isSealed, bool
isStatic, string eventName, IMethodSymbol invokeMethod, int parameterIndex)
        {
            var node =
SyntaxFactory.MethodDeclaration(SyntaxFactory.PredefinedType(SyntaxFactory.To
ken(SyntaxKind.VoidKeyword)),
SyntaxFactory.Identifier(GetEventMethodName(eventName)));
            if (isStatic)
            {
                node =
node.WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.Sta
ticKeyword)));
            }
            else if (isSealed)
            {
```

```
            }
            else
            {
                node =
node.WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.Pro
tectedKeyword), SyntaxFactory.Token(SyntaxKind.VirtualKeyword)));
            }
            node =
node.WithParameterList(SyntaxFactory.ParameterList(SyntaxFactory.SeparatedLis
t<ParameterSyntax>(new[] {
                SyntaxFactory.Parameter (SyntaxFactory.Identifier
(invokeMethod.Parameters [parameterIndex].Name)).WithType
(invokeMethod.Parameters [parameterIndex].Type.GenerateTypeSyntax ())
            })));
            return node;
        }

        static ExpressionSyntax GetTargetExpression(string declaringTypeName,
bool isStatic, string eventName)
        {
            if (eventName != "e")
                return
(ExpressionSyntax)SyntaxFactory.IdentifierName(eventName);

            if (isStatic)
                return
SyntaxFactory.MemberAccessExpression(SyntaxKind.SimpleMemberAccessExpression,
SyntaxFactory.IdentifierName(declaringTypeName),
SyntaxFactory.IdentifierName(eventName));

            return
SyntaxFactory.MemberAccessExpression(SyntaxKind.SimpleMemberAccessExpression,
SyntaxFactory.ThisExpression(), SyntaxFactory.IdentifierName(eventName));
        }

        static IEnumerable<ArgumentSyntax> GetInvokeArguments(bool isStatic,
ImmutableArray<IParameterSymbol> parameters, int parameterIndex)
        {
            if (parameters.Length > 1)
            {
                yield return SyntaxFactory.Argument(isStatic ?
(ExpressionSyntax)SyntaxFactory.LiteralExpression(SyntaxKind.NullLiteralExpre
ssion) : SyntaxFactory.ThisExpression());
            }

            yield return
SyntaxFactory.Argument(SyntaxFactory.IdentifierName(parameters[parameterIndex
].Name));
        }
```

```
        static MethodDeclarationSyntax CreateEventInvocator(string
declaringTypeName, bool isSealed, bool isStatic, string eventName,
IMethodSymbol invokeMethod, bool useExplictType)
        {
            int parameterIndex = (invokeMethod.Parameters.Length) > 1 ? 1 :
0;
            var result = CreateMethodStub(isSealed, isStatic, eventName,
invokeMethod, parameterIndex);
            var targetExpr = GetTargetExpression(declaringTypeName, isStatic,
eventName);
            result = result.WithBody(SyntaxFactory.Block(
                SyntaxFactory.ExpressionStatement(
                    SyntaxFactory.InvocationExpression(
                        SyntaxFactory.ConditionalAccessExpression(
                            targetExpr,

SyntaxFactory.MemberBindingExpression(SyntaxFactory.IdentifierName("Invoke"))
),
                        SyntaxFactory.ArgumentList(

SyntaxFactory.SeparatedList<ArgumentSyntax>(GetInvokeArguments(isStatic,
invokeMethod.Parameters, parameterIndex))))
            )));

            return result;
        }


        static MethodDeclarationSyntax CreateOldEventInvocator(string
declaringTypeName, bool isSealed, bool isStatic, string eventName,
IMethodSymbol invokeMethod, bool useExplictType)
        {
            int parameterIndex = (invokeMethod.Parameters.Length) > 1 ? 1 :
0;
            var result = CreateMethodStub(isSealed, isStatic, eventName,
invokeMethod, parameterIndex);
            const string handlerName = "handler";
            var targetExpr = GetTargetExpression(declaringTypeName, isStatic,
eventName);
            result = result.WithBody(SyntaxFactory.Block(
                SyntaxFactory.LocalDeclarationStatement(
                    SyntaxFactory.VariableDeclaration(
                        SyntaxFactory.ParseTypeName("var"),

SyntaxFactory.SeparatedList<VariableDeclaratorSyntax>(new[] {
                            SyntaxFactory.VariableDeclarator
(SyntaxFactory.Identifier (handlerName)).WithInitializer (
                                SyntaxFactory.EqualsValueClause (
                                    targetExpr
                                )
                            )
```

```
                        })
                    )
                ),
                SyntaxFactory.IfStatement(
                    SyntaxFactory.BinaryExpression(
                        SyntaxKind.NotEqualsExpression,
                        SyntaxFactory.IdentifierName(handlerName),

SyntaxFactory.LiteralExpression(SyntaxKind.NullLiteralExpression)
                    ),
                    SyntaxFactory.ExpressionStatement(
                        SyntaxFactory.InvocationExpression(
                            SyntaxFactory.IdentifierName(handlerName),
                            SyntaxFactory.ArgumentList(

SyntaxFactory.SeparatedList<ArgumentSyntax>(GetInvokeArguments(isStatic,
invokeMethod.Parameters, parameterIndex))
                            )
                        )
                    )
                ));
                return result;
            }

            static string GetEventMethodName(string eventName)
            {
                return NameProposalService.GetNameProposal("On" +
char.ToUpper(eventName[0]) + eventName.Substring(1),
SyntaxKind.MethodDeclaration);
            }
        }
    }
```

ДОДАТОК В

Компонент рефакторингу в інтегрованому середовищі розробки Visual Studio

Опис програми

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТР5166_19Б 13-1

Аркушів 8

Київ 2019

Анотація

Розділ містить опис частини, а саме рефакторинг провайдеру, яка слугує для автоматизації рефакторингу, що є атомарною структурною одиницею програмного продукту, та забезпечує виконання поставлених перед системою завдань. Призначенням провайдеру є аналіз блоку коду, та пропозиція можливих рішень по рефакторингу. Модуль надає можливість оптимізувати існуючий код або генерувати новий. Модуль написано мовою програмування C#, з використанням платформи компіляторів .Net.

ЗМІСТ

ЗАГАЛЬНІ ВІДОМОСТІ

У додатку розглядається один з програмних модулів системи — модуль для роботи з БД з кодом УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТР5166_19Б 12-1, що міститься у файлі AddNullCheckProvider.cs. Модуль реалізовано з використанням платформи Roslyn. Модуль призначений для створення рефакторингу, який буде інтегрований в IDE Visual Studio. Користувач має можливість отримувати для блоку коду пропозиції по рефакторингу та застосовувати їх без зайвих труднощів.

ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Призначенням модулю для AddNullCheckProvider є аналіз вхідного коду, та безпосередньо створення обгортки над існуючим оператором у вигляді оператору умови з перевіркою на значення null. Використання такого рефакторинг провайдеру дозволяє розширити Quick Actions меню в Visual Studio, надати користувачу можливість створення безпечного коду, шляхом передбачення виключення NullReferenceException, що може виникнути на етапі виконання.

ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Аналізувати код та пропонувати рефакторинг якщо він актуальний до заданого коду є головним завданням модуля. При запуску модуль отримує з контексту інформацію про документ з кодом, який відкритий в IDE Visual Studio, та аналізує чи підпадає вхідний код рефакторингу який реалізований в провайдері. Також модуль генерує новий документ який містить попередню структури, додаючи нові синтаксичну конструкцію, а саме оператор умови, навколо потенційно небезпечного коду.

ТЕХНІЧНІ ЗАСОБИ ЩО ВИКОРИСТОВУЮТЬСЯ

Модуль розроблено у середовищі розробки Microsoft Visual Studio 2017, що забезпечує набір сервісних функцій та графічний діалог з користувачем, на комп’ютері, що використовував операційну систему Windows 10. Модуль розроблений з використанням відкритих API що надаються платформою компіляторів .NET (“Roslyn”) SDK.

ВИКЛИК І ЗАВАНТАЖЕННЯ

Програмний модуль реалізований як окремий клас, який є атомарною одиницею. Оскільки клас має атрибути які сигналізують про те, що це рефакторинг який відображається в Quick Actions меню, то запуск відбувається під керуванням середовища розробки, без додаткових дій для користувача.

ВХІДНІ І ВИХІДНІ ДАНІ

Вхідними даними для модуля є програмний код.

Вихідними даними програмного модуля є новий пункт до Quick Actions меню, який додає перевірку на значення null.